E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

LoRaChat

Grado en Ingeniería Informática

Trabajo Fin de Grado

Ioar Crespo Diaz de Cerio
Francisco Javier Falcone Lanas
Adrián Catón Oteiza
Pamplona, junio de 2020

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Agradecimientos

Quiero agradecer enormemente a Francisco y Adrián por su ayuda en el terreno de las telecomunicaciones, en el que no había entrado antes como informático. Han sido de gran ayuda en este abrupto y *vírico* viaje, no solo en relación al Trabajo, sino también en relación a mi incursión en este mundo en calidad de radioaficionado amateur.

También quiero agradecer el apoyo de familia y amigos/as que me ha sido de mucha ayuda, sobre todo por la excepcional situación de crisis que la humanidad ha vivido y sigue viviendo estos últimos meses, en la que no ha sido fácil concentrarse ni llevar adelante el trabajo en condiciones óptimas, tanto psicológicas como experimentales.

Abstract

Information Technologies have brought us the possibility of living in a continuously and totally connected world. However, there are certain situations in which the flow of information is suddenly interrupted by certain events, such as natural disasters, political decisions, accidents in places where there is no coverage or other types of critical situations where communication infrastructures are affected or not reached, preventing normal operation of telematic networks. Faced with these possible scenarios, LoRaChat has been developed as a solution implementing an easy-to-deploy device and a secure communication protocol to hold long-distance conversations over LoRa radio modulation and over a mesh network created by user terminals themselves.

Laburpena

Informazio Teknologiek denbora osoan guztiz konektaturik dagoen mundu batean bizitzea ahalbidetu gaituzte. Hala ere, badaude egoera batzuk non informazio fluxu normala eragozten den, desastre naturalak, erabaki politikoak, koberturarik gabeko lekuetan gertatutako istripuak edo komunikazio infraestrukturak kaltetuak suertatzen diren eta haietara iristeko arazoak dauden bestelako egoera kritikoak direla medio, non sare telematikoen funtzionamendu normala eragozten den. Egoera posible hauen aurrean, LoRa irrati modulazioaren bidez eta erabiltzaile dispositiboei esker sortutako sare interkonektatu baten bidez distantzia handietan zehar elkarizketak edukitzea ahalbidetzen duen LoRaChat izeneko hedatze erraza duen dispositibo eta protokolo seguruak sortu dira.

Resumen

Las Tecnologías de la Información nos han brindado la posibilidad de vivir en un mundo totalmente conectado todo el tiempo. No obstante, hay situaciones en las que el normal flujo de información se ve súbitamente interrumpido por ciertos eventos, tales como desastres naturales, decisiones políticas, accidentes en lugares donde no hay cobertura u otro tipo de situaciones críticas donde las infraestructuras de comunicación son afectadas o no pueden alcanzarse, evitando el normal funcionamiento de las redes telemáticas. Ante estos posibles escenarios, LoRaChat ha sido desarrollada como una solución que implementa un dispositivo de fácil despliegue y un protocolo de comunicación seguro para mantener conversaciones a largas distancias sobre la modulación de radio LoRa y sobre una red con topología de malla creada por los propios terminales de usuario.

Palabras clave

LoRaChat, LoRa, ESP32, radio, encriptación, chat, malla, red, protocolo, WiFi, WebSockets, paquete, routing, microcontrolador.

Índice

1. Introducción

El mundo de las telecomunicaciones y de las Tecnologías de la Información se ha desarrollado tanto en los últimos años que se nos hace difícil imaginarnos diferentes procesos (industriales, políticos, sociales, culturales, etc.) sin la intervención de estas tecnologías de comunicación. Sin embargo, hay momentos en los que, por determinadas causas y en determinadas circunstancias, el normal y correcto funcionamiento de estas vías de comunicación se ve mermado y el flujo de información se corta, causando todo tipo de estragos en los ámbitos en los que operan.

Efectivamente, ya sea por un desastre natural, una saturación de la red, un ataque a la comunicación o ya sea por algún tipo de decisión política, la comunicación que se sostiene sobre la base de estas redes es afectada o, en el peor de los casos, se interrumpe. Es en estos momentos cuando se plantea la necesidad de una solución para que el flujo de información continúe por otros cauces.

Cabría pensar que si en un punto geográfico concreto el acceso a Internet se ve interrumpido, la solución podría venir por parte de otras vías como, por ejemplo, la radio: analógico, DMR, Tetra, etc. Y viceversa: si éstas caen, la solución sería Internet. No obstante, estas infraestructuras disponen de una regulación y una serie de restricciones técnicas que predeterminan su uso y limitan el ámbito de operatividad. Fundamentalmente, esto ocurre porque la configuración de redes de este tipo precede al momento del colapso. Además, el despliegue de estas tecnologías, mediante repetidores, enlaces y terminales, no es precisamente barato. Incluso habría que tener en cuenta que durante o después de desastres naturales o similares, si el Internet se ha caído, seguramente lo hayan hecho también otros medios de telecomunicación.

Otro escenario típico podría ser la falta de cobertura por parte de sistemas de comunicación ya establecidos en el terreno. Por ejemplo, un accidente en un túnel o en un valle de una zona montañosa.

Es en este tipo de situaciones, donde la comunicación no puede establecerse sobre redes cotidianas, cuando se requiere una solución de rápido despliegue que ofrezca características como la facilidad de configuración y despliegue, la seguridad, la facilidad de uso y el bajo coste, entre otras.

2. Resumen y objetivos del proyecto

Para responder a la problemática expuesta en la Introducción, se ha planteado desarrollar una solución que de respuesta teórico-práctica a la necesidad de vías de comunicación en ese tipo de situaciones.

A la solución propuesta se la ha llamado LoRaChat. El nombre describe las dos características principales de la solución: un chat de texto sobre la modulación de radio LoRa.

Su intención concreta es la de ofrecer comunicaciones de chat de largo alcance, mediante un único dispositivo hardware que integre tanto la interfaz de usuario, como la capa de transporte de la red, cuya topología toma la forma de malla, pues es la que se ha concluido como la más acorde para la solución.

Para ello, se han desarrollado un protocolo de red, un sistema de encriptación, una interfaz de usuario y un dispositivo hardware que, además de realizar las anteriores acciones y las relativas al usuario, actúa como nodo y rúter de una red de comunicación mallada.

3. Estado del Arte

Existe todo un mundo de soluciones para las situaciones descritas en la introducción, es decir, situaciones en las que el acceso a Internet por las vías convencionales no está disponible o, en algún caso, no es deseable. De todas estas soluciones, las que más interesan al proyecto son aquellas cuyo objetivo es proporcionar la capacidad de chatear sobre redes ajenas a Internet.

Una de las aplicaciones más conocidas para este propósito es Serval Project **[1]**. Este proyecto, mediante sus aplicaciones para móviles, creaba una red en malla entre diferentes terminales a través de WiFi y proveía una serie de servicios sobre esa red, entre los que se encontraba un chat encriptado. Sin embargo, el proyecto ha ido dejándose de lado con el tiempo.

Esta llamada a la creación de aplicaciones de este estilo fue recogida por diferentes iniciativas. Una de ellas fue Firechat **[2]**. Esta app, disponible para Android e iOS, creaba una red en malla entre terminales móviles, no solo a través de WiFi, sino también por medio de Bluetooth. Si uno de los dispositivos de la red estaba conectado a Internet, los demás dispositivos de esa misma red mallada también tendrían acceso a Internet, pero sólo para el uso del chat. No obstante, y una vez más, esta aplicación ha ido desactualizándose con el tiempo, hasta el punto de que ya no está publicada en las tiendas de Android ni de iOS.

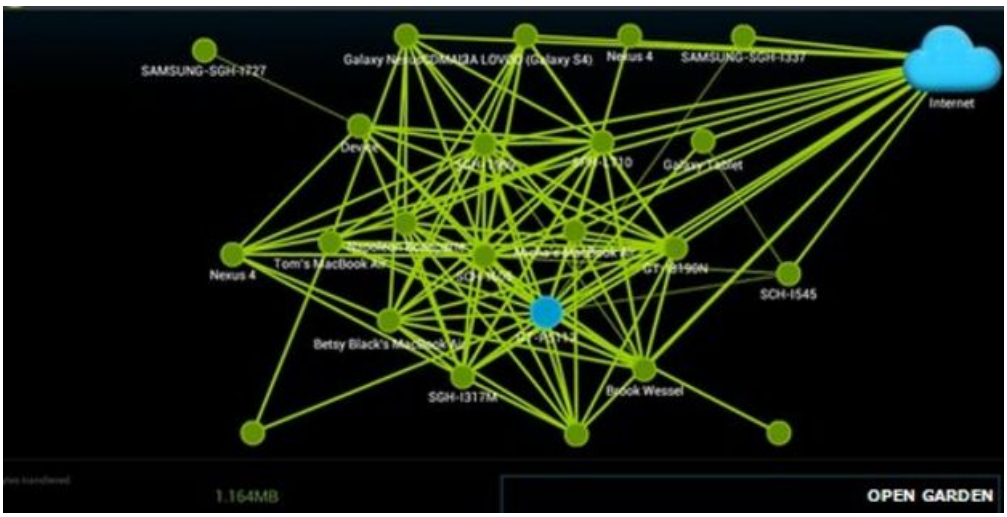


Fig. 1 Malla creada con Firechat. Fuente: https://ichef.bbci.co.uk/news/624/media/images/74553000/jpg/_74553691_garden2.jpg

Otra de las aplicaciones que recogió este testigo es Briar **[3]**. A diferencia de las anteriores, Briar sigue en desarrollo y en constante actualización, lo que la hace una de las aplicaciones de referencia en este ámbito. Al igual que Firechat, Briar permite la comunicación sobre WiFi, Bluetooth y, cuando lo hay, Internet. Todo ello sobre una red

con topología de malla. También permite la utilización de Tor para hacer anónimo y encriptar el tráfico (aunque los mensajes ya están por defecto encriptados). Para el transporte, utiliza un protocolo llamado Bramble Transport Protocol (BTP) **[4]**.

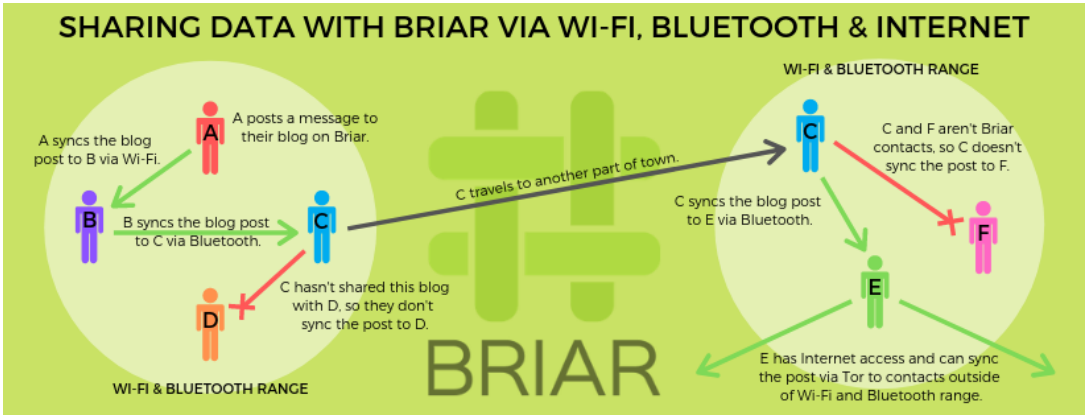


Fig. 2 Funcionamiento de Briar. Fuente: https://www.bbc.com/news/technology-27225869

No obstante, todas estas aplicaciones, que de una u otra manera solucionarían el problema planteado en este trabajo, tienen una gran limitación: la distancia. Efectivamente, la proximidad entre terminales es de vital importancia para su funcionamiento, pues tecnologías como WiFi y Bluetooth, al menos del modo en que vienen integrados en los dispositivos de consumo habitual (smartphones y ordenadores), tienen un alcance de entre 10 y 100 metros. Por ello, se hace necesario buscar soluciones en otro tipo de capa física.

Cuando se trata de comunicaciones inalámbricas de largo alcance y con un nivel de infraestructura simple, una de las tecnologías que suele venir a la mente en primer lugar es LoRa. Esta modulación de radio, cuyo funcionamiento y características se analizarán en apartados posteriores, permite comunicaciones inalámbricas a gran distancia (10 Km y hasta 200 Km con configuraciones hardware específicas) con un consumo de energía relativamente bajo y una configuración hardware sencilla. Por tanto, desde que LoRa se presentó en el mercado hace poco más de media década, no se han hecho de esperar las soluciones de mensajería aplicadas sobre esta capa física.

Durante la investigación de soluciones ya desarrolladas sobre la temática chat y LoRa, se han encontrado diversas implementaciones, aunque la mayoría complejas de cara al usuario inexperto. Una de esas soluciones es Ripple LoRa Mesh **[5]**. Este es un proyecto que puede encontrarse en GitHub **[6]**, pero, por desgracia, su código no está abierto, por lo que únicamente pueden encontrarse los binarios para cargar en la placa. Esto hace que no se haya podido analizar su código ni el protocolo que implementa, como base para el desarrollo de una solución propia. El proyecto necesita de una placa de desarrollo ESP32 o Sparkfun para actuar como enlace LoRa y un smartphone con sistema operativo Android para la ejecución de la aplicación cliente de chat Ripple Messenger. La conexión entre la placa y el smartphone se establecería por medio de un cable USB

OTG. La principal limitación de este proyecto podría decirse que es la necesidad de una aplicación cliente específica, así como su correlativa dependencia del sistema operativo o la plataforma de ejecución.

Un estudiante de ingeniería electrónica de India también ha desarrollado un dispositivo parecido, con el objetivo de chatear usando LoRa **[7]**. Esta vez, su proyecto permite que el cliente se ejecute tanto en smartphone como en una computadora. Además, el funcionamiento del proyecto está debidamente explicado (aunque no formalmente detallado). Para el envío y recepción de datos por LoRa, el autor ha diseñado una placa PCB en la que se integrarían una placa ESP8266 y un módulo LoRa RYLR896. En esta placa se cargaría el código disponible en GitHub, cuya única función es enviar y recibir una serie de comandos AT para hacer funcionar el módulo RYLR896. Los mensajes para enviar se comunicarían en texto plano a través de una conexión serial, para ser recogidas por el ESP8266, crear el comando AT y enviarlo al módulo LoRa. Finalmente, el módulo LoRa enviaría el mensaje por radio. Para recibir, ocurriría el mismo proceso, pero a la inversa. El problema más grande de esta implementación reside en que no hay un cliente amigable para el usuario, aunque su uso se de mediante cualquier terminal capaz de comunicarse por serial, lo que hace que sea, de cierta manera, cross-platform. Otro problema, no menos importante, es que no implementa ningún tipo de encriptación para los mensajes.

Otro proyecto muy interesante y más complejo puede encontrarse en este **[8]** repositorio de GitHub. Aunque no es un desarrollo terminado, es un compendio de ideas sobre un desarrollo en curso. Lo que trae de nuevo este prototipo es su interoperabilidad entre la red LoRa e Internet. La forma en que este proyecto funcionaría sería por medio de un Raspberry Pi que alojaría un servidor XMPP y se conectaría haciendo uso del protocolo de enlace AX25 sobre LoRa (ejecutado por un Arduino). Tal y como lo resume el autor: Raspberry Pi - Arduino - LoRa - AX25 - XMPP. Sobre XMPP **[9]** pueden realizarse muchas acciones más aparte de chatear, además de contener una gran variedad de características, entre las que se encuentra la capacidad de encriptar los mensajes mediante protocolos como OTR u OMEMO. No obstante, y como se ha dicho, es un proyecto que no está terminado y que ni siquiera tiene una primera versión disponible.

Por último, el proyecto de mensajería sobre LoRa que más se asemeja a la solución que se va a describir en este trabajo, es uno que ha implementado un servidor web local en un ESP32 para servir el cliente de chat **[10]**. La forma de funcionar del sistema es similar a los proyectos anteriormente descritos, donde una placa (ESP32) utiliza un módulo LoRa (Ra-02) para enviar mensajes por radio. La novedad reside, pues, en que al ser el cliente una interfaz web, permite que cualquier dispositivo con un navegador y la capacidad de conectarse a una red WiFi, pueda usar el sistema.

4. Fundamentos teóricos

4.1 LoRa

LoRa (Long Range) se define como una capa física o modulación inalámbrica (radio) para crear un enlace de comunicación de largo alcance **[11]**. La modulación y el protocolo de LoRa fue diseñado por Semtech con el objetivo de permitir comunicaciones de largo alcance con un consumo de energía mínimo. Esto permite a los chips que implementan LoRa enviar pequeñas cantidades datos a más de 10 km en áreas rurales, sobre frecuencias de uso libre (sin licencia), como son la 433 MHz y la 868 MHz en Europa, la 915 MHz en Norte América y Australia, y la 923 MHz en Asia.

Como se ve, las principales ventajas frente a otras modulaciones o, dicho de otra manera, las características distintivas de LoRa con respecto a tecnologías como WiFi, Bluetooth, SigFox, ZigBee y otras, son el largo alcance y el bajo consumo de energía, con buena solidez ante interferencias y una alta sensibilidad para recibir datos (-168dB). Sin embargo, detenta un ancho de banda bajo-moderado, debiendo enviarse pocos datos en el tiempo.

La modulación LoRa utiliza una modulación de amplio espectro derivada de la modulación Chirp Spread Spectrum (CSS). Ésta última es una técnica de amplio espectro que utiliza chirping (frecuencia modulada pulsada) de ancho de banda linear para codificar la información. En LoRa esto se ejecuta de tal manera que cada bit de información transmitida es representado por varias pulsaciones llamadas chirps **[12]**.

Si bien se ha mencionada la capa física de LoRa, también existe una capa de red (y superiores) estandarizada para el uso de la tecnología LoRa en aplicaciones de IoT. El protocolo más conocido, que implementa lo descrito anteriormente es LoRaWAN (LoRa Wide Area Network), perteneciente a LoRa Alliance **[13]**. LoRaWAN define un protocolo de comunicación y arquitectura de sistema que permite la interconexión de diferentes dispositivos de IoT que implementan LoRa a un gateway que los conecta a internet y a la nube. La topología de esta red es de estrella, encargándose el gateway de centralizar la gestión de acceso a la red **[14]**.

4.2 Topología de red

Con esta información, se comenzó a plantear la solución concreta a la problemática planteada. Uno de esos primeros retos fue definir la topología de red que representaría LoRaChat.

Existiendo LoRaWAN, una de esas primeras ideas fue implementar una topología de estrella, donde el nodo central y centralizador fuese un gateway de LoRaWAN. Sin embargo, esto planteaba una serie de problemas. El primero era que el gateway LoRaWAN está principalmente pensado para integrar los dispositivos IoT con la nube e Internet. Pero si se partía de la existente probabilidad de que, si va a usarse LoRaChat,

aparte de que seguramente el acceso a Internet no esté disponible, tampoco pueda asegurarse que haya acceso a las funcionalidades de un gateway LoRaWAN, la idea no era muy convincente. También es probable que, si no hay acceso a Internet o a otros medios que utilizan el aire, como DMR o Tetra, la cobertura tampoco sea suficiente para alcanzar una gateway LoRaWAN. El segundo problema que se planteó es que quedaba muy restringido el uso de LoRaChat si se hacía dependiente de un nodo central. En caso de que el nodo cayese o no estuviese desplegado, cosa muy probable durante un desastre natural o una situación que no fuese predicha con antelación, la comunicación por LoRaChat quedaba obsoleta.

Ante esta serie de problemáticas, se pensó que lo más adecuado sería implementar una topología de red en forma de malla. De este modo, la disponibilidad de la comunicación sería más resistente y adaptable ante la caída de nodos de la red. Si cae un nodo, no tiene por qué caerse la red. Además, facilitaría el despliegue, de tal forma que bastaría con encender los dispositivos en la situación requerida para crear la red y comunicarse sobre ella. Fue así acordada la topología en forma de malla.

No obstante, antes de la solución definitiva, la idea de malla se había propuesto de forma híbrida: unos rúters intermedios jugarían el papel de crear la red mallada, mientras que los terminales de usuario utilizarían esa red. Aunque esto seguía teniendo ventajas ante la dependencia de un nodo central, dificultaba considerablemente el despliegue, ya que se requería el despliegue de una serie rúters para poder usar la red. Así que, definitivamente, se decidió fusionar las funcionalidades de todos los nodos de la red, creando, por consiguiente, una red con topología de malla pura: los terminales de usuario actuarían a su vez como rúters. Fue así como se simplificó el desarrollo de la solución, debiendo programarse un único tipo de dispositivo que cumpliría todas las funciones de LoRaChat.

Una red con topología en malla se define como un conjunto de nodos de red en el que cada nodo está conectado a todos los demás nodos. Este tipo de topología de red, a diferencia de otras topologías como la de estrella, no requiere de un servidor central para su funcionamiento, lo que la convierte en una candidata perfecta para la solución buscada. Además, es una red muy tolerante a fallos, pues la caída de uno o más nodos o de uno o más enlaces, no supone necesariamente que la comunicación se vea interrumpida. Los nodos adyacentes a la caída pueden notificar a los demás las nuevas rutas a seguir, siempre y cuando el protocolo de enrutamiento esté diseñado para ello. Los dispositivos que normalmente se conectan a una red mallada suelen ser clientes, rúters y gateways.

Las redes en malla requieren de unos protocolos de enrutamiento diferentes a los habituales que se dividen en tres tipos: proactivo, reactivo e híbrido **[15]**. Cada protocolo hace variar los resultados en cuanto a escalabilidad y rendimiento.

Un protocolo proactivo es aquel que está en continuo modo de descubrimiento. Cuando ocurre un cambio en la red, sus nodos informan unos a otros sobre las rutas existentes y comparten la información que cada cual tiene en su tabla de rutas. De este modo, se habilita a cada nodo para ser capaz de escoger la ruta óptima y mandar los paquetes por

el mejor enlace. También es útil ante caídas de nodos o enlaces, pues se obliga a actualizar las tablas de rutas y recalcular las mejores rutas para alcanzar uno u otro nodo. Este es un tipo de protocolo adecuado para redes estáticas o en las que rara vez ocurren cambios o caídas en los enlaces, ofreciendo un buen tiempo de respuesta ante estas situaciones, mejorando la tolerancia a fallos de la red. Sin embargo, en redes no tan estáticas, donde a menudo se rompen enlaces o se caen nodos, este protocolo no ofrece tan buen rendimiento, porque cuanto más crece la inestabilidad de la red, tanto más aumenta la carga en la red, pues aumenta el tráfico (de paquetes de configuración), incrementando la probabilidad de colisiones y ralentizando la red en la misma proporción.

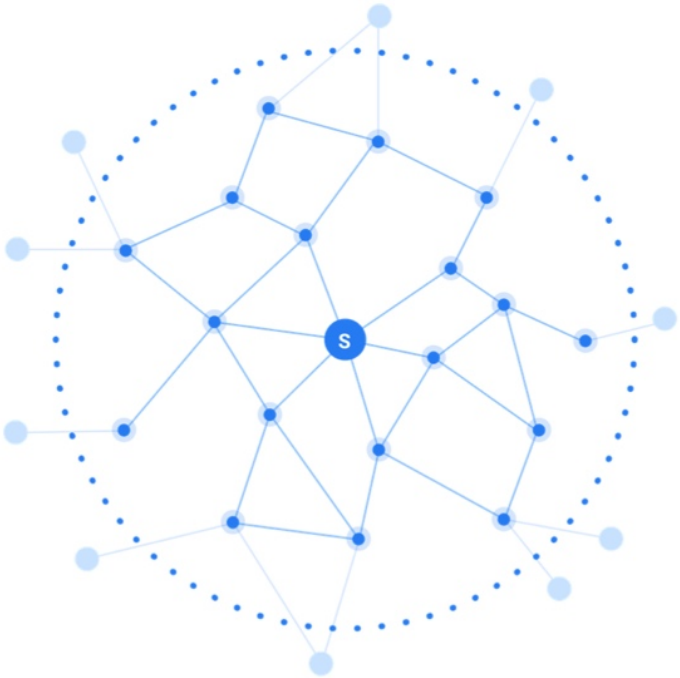


*Fig. 3 Protocolo proactivo. Fuente: https://hackernoon.com/hn-images/1*0NuZcB-R4PV0xXp25Ph7ZQ.png*

Por su parte, un protocolo reactivo establece rutas bajo demanda. Es decir, cada nodo tiene que pedir a toda la red la ruta óptima para establecer una conexión. Esto evita que continuamente se envíen mensajes de ajuste y solo se haga cuando se requiere. De este modo, este tipo de red escala mejor, pero tarda más tiempo en establecer conexiones, precisamente porque requiere que la red le proporcione la mejor ruta.

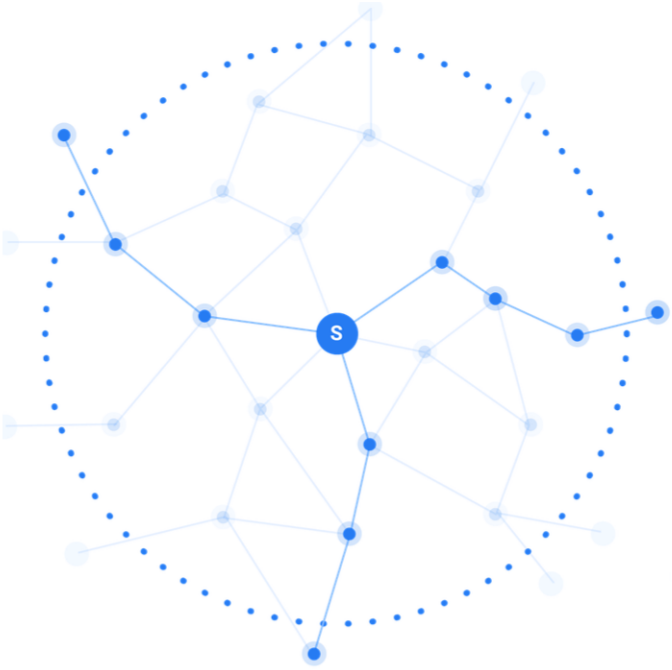


*Fig. 4 Protocolo reactivo. Fuente: https://hackernoon.com/hn-images/1*pGYSApK8Rpx5XDCORePnGA.png*

Un protocolo híbrido se usa cuando ni el proactivo ni el reactivo se adecuan a la red objetivo. Suelen usarse en casos específicos, ad hoc.

En cuanto a la topología en malla, esta puede ser completa o parcial. El nombre está directamente relacionado con la teoría de grafos. Si una red en malla es un grafo completo, es decir, donde todos sus vértices (nodos) están conectados entre sí mediante aristas (enlaces), entonces se tiene una malla completa. Si esta condición no se cumple, la malla es parcial.

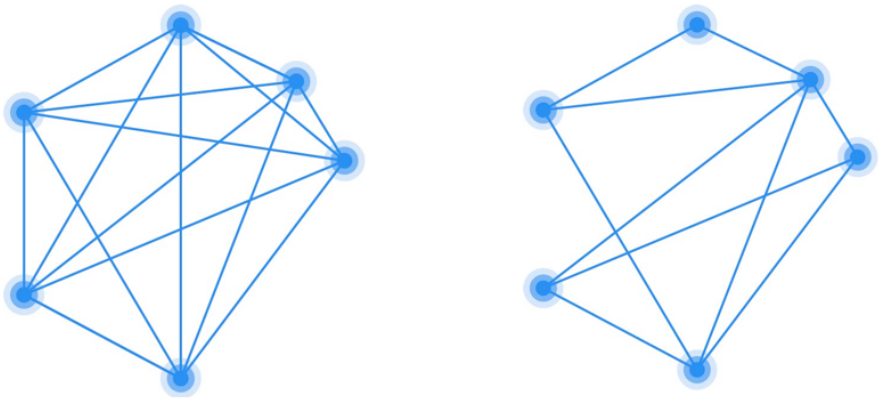


*Fig. 5 Redes en malla completa y parcial. Fuente: https://hackernoon.com/hn-images/1*ejwFszve3WjAi49Hmb48SA.png*

Las redes en malla también pueden desplegarse sobre diferentes tipos de transporte. Por ejemplo, parte de la red podría estar conectada vía cable, mientras otra parte podría estar conectada por radio y funcionar ambas abstrayéndose totalmente del medio físico en el que se desenvuelven. Normalmente las redes malladas están compuestas por enlaces de ambos tipos.

En el caso del problema que se quiere resolver, la red se va a desplegar sobre el aire. Es decir, que la conexión entre nodos se va a hacer por radio (ondas electromagnéticas). Esto requiere una breve introducción a las Redes Inalámbricas Malladas (WMN) y sus principales protocolos.

Una WMN no es más que una red mallada que opera sobre un medio inalámbrico (radio). Al ser un único medio el que se comparte por todos los nodos (a diferencia de redes en las que el medio es un cable), hay que tomar algunas medidas para evitar las colisiones entre las comunicaciones de los diferentes nodos. Una de las maneras más típicas de hacer frente a esta necesidad es utilizando canales entre un rango de frecuencias, en caso de la utilización de una modulación de frecuencia (FM). Esta solución, por tanto, asigna un canal a cada enlace, de tal modo que la comunicación entre dos nodos se lleve a cabo por un canal diferente al de los nodos cercanos. Si bien el espacio de frecuencias es limitado, normalmente por la regulación del espacio de radiofrecuencia o por la utilización que de él hacen otros servicios cercanos, se trata precisamente de repartir un conjunto de canales con la mira puesta en su reutilización allí donde no colisione con otro enlace.

Otra forma de hacer frente a las posibles colisiones es utilizando técnicas de multiplexación de un canal (TDM, Time división multiplexing). Uno de los métodos más comunes es TDMA (Time-division multiple access), donde un canal se divide en varios timeslots, utilizado por GSM (8 timeslots, 4.615 ms frame duration), DMR (2 timeslots, canal de 12.5 kHz), TETRA (4 timeslots, canal de 25 kHz) **[16]**.

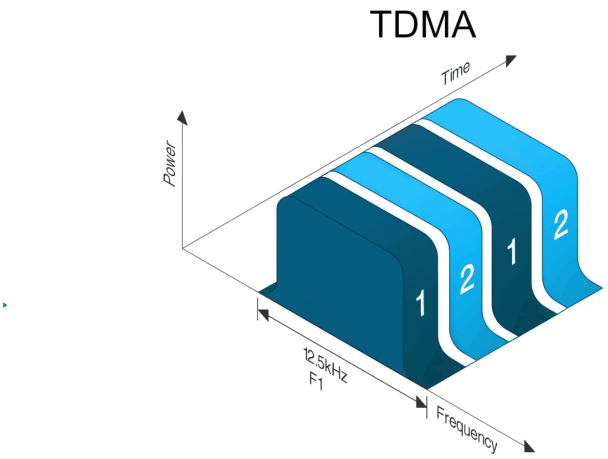


Fig. 6 Gráfico TDMA. Fuente: https://www.taitradioacademy.com/wp-content/uploads/2014/12/Image-21.png

En cuanto a protocolos de enrutamiento para redes con topología de malla, existe una gran variedad. De todos ellos se van a describir dos de los más usados: B.A.T.M.A.N. Adv. y HWMP.

B.A.T.M.A.N. Adv. es una extensión del protocolo B.A.T.M.A.N., que principalmente se diferencia de este por funcionar a nivel de enlace en vez de funcionar en la capa IP y en el espacio de usuario dentro del sistema operativo **[17]**. Esta diferencia le permite a B.A.T.M.A.N. Adv. ahorrar espacio y funcionar de manera más rápida. El protocolo dispone de seis paquetes diferentes: paquete batman, ICMP, paquete unicast, paquete unicast fragmentado, paquete broadcast y paquete de visualización. El más importante de estos es el paquete batman, ya que el resto sirven, en general, para proveer servicios o extensiones sobre el protocolo. Un paquete batman, también conocido como Originator Message (OGM), sirve para para descubrir nodos y rutas en la red, así como para medir la calidad de la red, y provee la siguiente información:

- Dirección del nodo que ha generado el paquete.
- Número de secuencia. Sirve para determinar paquetes duplicados y para medir la calidad del enlace.
- Calidad de transmisión o Transmit Quality (TQ). Describe la calidad de la ruta total hasta el nodo originador.
- Dirección del anterior emisor. Sirve para descartar OGMs recibidos previamente por el receptor.
- Time To Live (TTL). Limita el número de nodos por los que puede pasar un paquete batman.
- Contador de Host Neighbour Announcement (HNA). Es decir, un contador del número de nodos que no implementan batman al que puede llegar el originador del mensaje.
- Flags de gateway. Se usan cuando el nodo que ha generado el paquete ofrece conexión a otra red.

Al ser un protocolo proactivo, los nodos envían mensajes OGM de forma periódica a sus vecinos. Cuando un vecino recibe un paquete OGM, realiza las siguientes acciones (aparte de otras para evitar bucles y paquetes duplicados):

1. Comprobar si el origen del OGM es el propio nodo. En este caso, el emisor es un vecino directo. Se actualiza la tabla de rutas.
2. Comprobar si el emisor del OGM anterior es el propio nodo. En tal caso, el paquete ya fue analizado y, por tanto, se desecha.
3. Determinar el origen del OGM y si no existe en la tabla de rutas, crear la entrada.
4. Actualizar la calidad del enlace hasta el origen.
5. Actualizar los campos TQ y TTL del paquete y retransmitirlo de nuevo en broadcast.

La calidad de transmisión o TQ indica la probabilidad de un paquete para llegar a su destino. Para su cálculo se usan dos métricas, Receive Quality (RQ, la probabilidad de

recibir paquetes desde el nodo destino al nodo origen) y Echo Quality (EQ, la probabilidad de que el nodo origen reciba de vuelta el paquete que envió al nodo destino), agregándolas según la fórmula:

$$TQ = EQ/RT$$

Este valor se va actualizando durante la propagación de la siguiente manera:

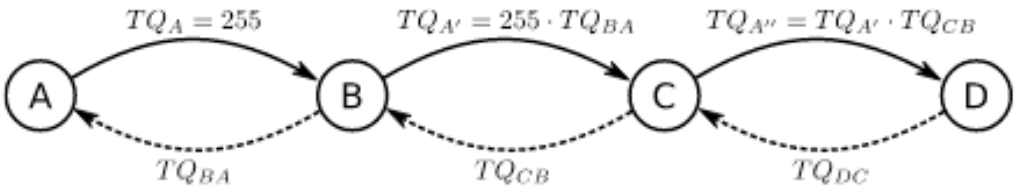


Fig. 7 Proceso de actualización de la métrica de calidad en B.A.T.M.A.N. Adv. Fuente: https://downloads.open-mesh.org/batman/papers/batman-adv_network_coding.pdf

A la hora de escoger la ruta, solo se tiene en cuenta el siguiente salto. Es decir, en vez de una visión global de la red, se decide por dónde enviar únicamente en base a qué enlace tiene la mejor métrica.

El otro protocolo analizado es Hybrid Wireless Mesh Protocol (HWMP) **[18]**. Tal y como indica su nombre este es un protocolo híbrido, ya que combina el enrutamiento bajo demanda con extensiones proactivas de árbol de topología. Se basa parcialmente en el protocolo Ad Hoc On Demand (AODV). Utiliza cuatro tipos de mensaje de control: Route Request (RREQ), Route Reply (RREP), Route Error (RERR) y Root Announcement (RANN). Para el cálculo de las rutas, se utiliza métrica, incluida en los mensajes RREQ, RREP y RANN. Al igual que en B.A.T.M.A.N. Adv., se utilizan números de secuencia para evitar bucles en la red.

En el modo bajo demanda (reactivo), HWMP funciona enviando un RREQ para solicitar la ruta hasta un destino en concreto. Cuando un nodo recibe un RREQ, este crea una ruta en caso de que la métrica que contiene el RREQ sea peor que la que el nodo tiene almacenada. En caso contrario, el nodo actualizará su información en base a la métrica del RREQ. En caso de crear una nueva ruta o modificar una existente, el RREQ se reenvía. El mensaje RREQ se va actualizando de forma acumulativa en lo que a su métrica se refiere. Finalmente, cuando el RREQ alcanza el nodo destino, éste envía un RREP con la información de la ruta, encaminado hacia el nodo origen. Existe un flag que puede definir si los RREP solo los enviará el destino o también pueden hacerlo los nodos intermedios que conozcan una ruta hasta ese destino.

En el modo proactivo de construcción de árbol, existen dos métodos. Uno utiliza RREQ proactivos para generar rutas entre el nodo raíz y el resto de los nodos de la red. El otro método, utiliza un RANN con el objetivo de distribuir información de rutas para alcanzar

la raíz, aunque las rutas pueden crearse bajo demanda. En el primer método, el nodo raíz envía periódicamente mensajes RREQ con el flag de destino con valor broadcast y, de manera similar al modo bajo demanda, el árbol se va creando poco a poco. En el segundo método, el nodo raíz envía periódicamente un mensaje RANN para diseminar las métricas de las rutas hasta la raíz entre todos los nodos.

4.3 Malla LoRaChat

Pese a lo instructivos que son los dos protocolos anteriores, se ha querido prescindir de una especificación previa. Esto atiende a un importante factor que influye directamente en la solución que se le ha querido dar a la problemática: el rápido despliegue. En tanto que esto es así, no se puede hacer depender un protocolo de IDs creados al arbitrio de los usuarios, ya que esto acarrearía una serie de problemas, como la duplicación de IDs. A diferencia de otros ámbitos, donde a nivel de enlace existen las direcciones MAC o a nivel de red existen las direcciones IP, en el ámbito para el cual se desarrolla LoRaChat no existe centralización en el reparto de direcciones. O bien se reparten con anterioridad, lo que haría de la solución un sistema inútil, pues ya existen tecnologías mucho más robustas que podrían suplir su papel (DMR, TETRA, etc.) o bien se requiere de un nodo central que reparta las direcciones, cayendo de nuevo en la dependencia de una topología centralizada. Además, los módulos LoRa no disponen de una dirección única de fábrica, sino que hay que configurarla a nivel de aplicación (LoRaWAN MAC). Entonces, la solución a todas estas problemáticas tiene que venir de un protocolo sin direcciones.

La forma en la que se ha logrado esto es utilizando identificadores de paquete. Éstos identifican de manera casi única cada paquete, lo que permite a los nodos de la red determinar si el paquete debe ser procesado o reenviado o, en cambio, ya ha sido procesado o reenviado con anterioridad. Esto supone, por consiguiente, que todos los nodos enruten el tráfico de forma ciega en cuanto a la red y los enlaces, con la única información de si el nodo ya ha recibido un paquete o no. Además del identificador de mensaje, la cabecera del paquete también incluye un identificador de grupo o talkgroup (TG), para que solo los nodos que pertenezcan a ese TG procesen el paquete. Aquellos nodos que no pertenecen al TG indicado en la cabecera del paquete, únicamente se limitarán a reenviarlo si éste no fue reenviado con anterioridad por ellos mismos. Para estos dos propósitos, cada nodo dispone de una tabla donde se almacenan los IDs de los paquetes recientes (tabla que tiene un límite de entradas y se rellena de forma circular) y otra tabla donde se almacenan los IDs de los TGs a los que el nodo está suscrito o, dicho de otra manera, aquellos TGs de los cuales se dispone de la clave de cifrado.

A nivel más elevado, estrictamente de aplicación, se utilizan identificadores de grupos, autores y mensajes, formando estos el payload o carga del paquete. Al ser un protocolo exclusivamente desarrollado para una aplicación, no ha sido pensado para extrapolarse a otros usos. No obstante, el estudio y aplicación del protocolo basado en identificador de paquetes pueden ser útiles para desarrollos futuros.

Una característica añadida del protocolo es que está pensado para ser un protocolo seguro desde el principio. Mediante el uso de hashes y protocolos de encriptación, se ha conseguido que el contenido de los paquetes sea privado, que los paquetes no se puedan falsificar, que se mantenga la integridad y, finalmente, que sea difícil determinar la estructura de la red, con el objetivo de evitar posibles ataques de denegación de servicio.

Todos los detalles relativos al protocolo quedan descritos en el capítulo 7, dedicado a la especificación del protocolo de comunicación, como son la constitución de los paquetes, las secuencias de envío, recepción y enrutamiento. Por ahora bastará con decir que existen dos tipos de paquetes: los paquetes que contienen un mensaje y los paquetes ACK, que confirman la recepción de un mensaje.

El siguiente escenario de una malla LoRaChat sirve para explicar cómo funciona la malla.

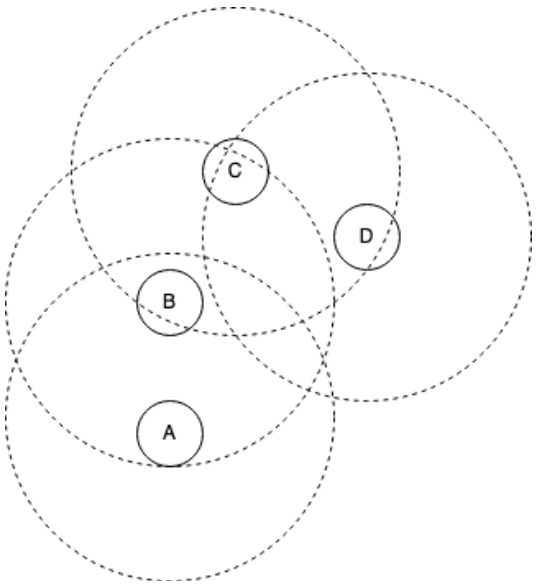


Fig. 8 Escenario para explicación de malla LoRaChat

En la anterior imagen, se pueden ver los rangos de alcance de cada nodo de la red.

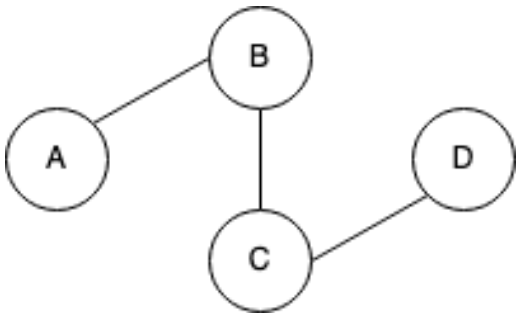


Fig. 9 Enlaces lógicos del escenario para la explicación de la malla LoRaChat

En esta otra, se aprecian los pseudo-enlaces que existirían en esa red. Según este escenario, el nodo A solo podría alcanzar al nodo B, y viceversa. El nodo B podría alcanzar al nodo C. El nodo C alcanzaría a C y D. Por último, el nodo C solo podría alcanzar a D.

Si, por ejemplo, el nodo A enviase un mensaje a determinado TG, al no tener éstos un identificador de destino, irían a parar a toda la red. Es decir, sería recibido por B que, a su vez, lo reenviaría a C y, de nuevo hacia A, aunque éste último lo descartaría por ser reciente en su tabla. C lo reenviaría a D y B, de los cuales este último lo rechazaría por ser reciente. Finalmente, D lo reenviaría a C, que también lo rechazaría. Cuando un nodo rechaza un mensaje por reciente, no lo reenvía. De este modo se evitan los bucles.

Solo aquellos nodos pertenecientes al TG indicado en la cabecera del paquete procesarían el paquete para extraer el contenido del mensaje. Estos enviarían un ACK que seguiría la misma lógica de enrutamiento descrita más arriba, en aras de informar al emisor de la recepción.

5. Herramientas utilizadas

En este apartado se pretende presentar las herramientas hardware y software que han sido utilizadas para desarrollar la aplicación.

5.1 Hardware

La única pieza de hardware que se ha utilizado para el desarrollo de LoRaChat ha sido un ESP32. Más en concreto, el ESP32 de TTGO que trae embebido un módulo de radio LoRa.

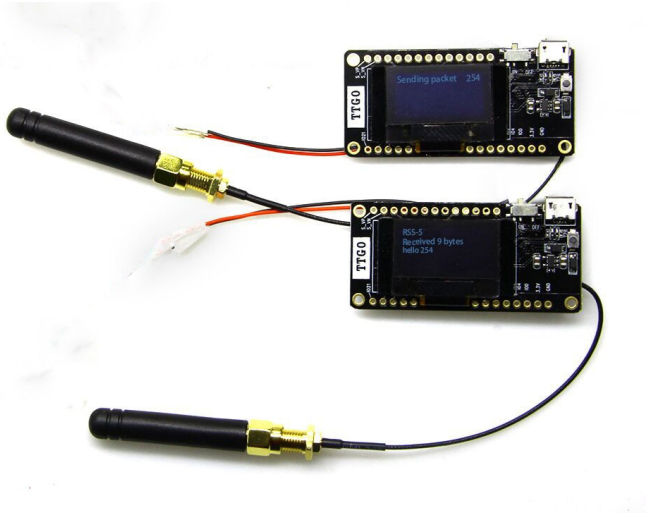


Fig. 10 Placas ESP32. Fuente: https://ae01.alicdn.com/kf/HTB1UILTkBfH8KJjy1Xbq6zLdXXar/2-Pcs-TTGO-LORA32-V2-0-868-433Mhz-ESP32-LoRa-OLED-0-96-Inch-SD-Card.jpg

El ESP32 es una placa o, más bien, una familia de placas SoC (sistema en chip, es decir, que integran todos los módulos necesarios para el funcionamiento de la placa en un único chip), desarrollado por Espressif Systems. A diferencia de otras soluciones del estilo, como podría ser la famosa familia de placas Arduino, basadas en microcontroladores AVR, la placa ESP32 dispone de varias características ventajosas como la inclusión por defecto de módulos embebidos de WiFi y de Bluetooth, un almacenamiento interno (SPIFFS) y un consumo energético muy reducido. Esto lo hace de gran utilidad en el mundo del Internet of Things (IoT).

Las características propias del ESP32 a la que más uso se le ha dado han sido la zona WiFi y el almacenamiento interno, con el objetivo de servir todos los recursos del sistema a partir de un único componente que actúa como centro.

Como se ha dicho más arriba, la placa concreta que se ha elegido es la TTGO ESP32 LoRa, porque ya trae embebido un módulo de radio LoRa, simplificando de manera considerable la solución, en cuanto a hardware se refiere. De este modo, tenemos una monolítica y compacta pieza de hardware sobre la que desarrollar toda la solución, en

vez de tener que conectar un módulo LoRa externo, debiendo estañar cables y dejando el entorno menos ameno. Es decir, el usuario solo necesitará una placa con el programa cargado (aparte de la fuente de alimentación, que podría ser una batería) y la pantalla para visualizar la interfaz gráfica.

El módulo LoRa que lleva integrado la placa es un SX1276 de Semtech **[19]** que opera a 868 MHz, aunque LoRaChat ha sido desarrollado para ser independiente de la frecuencia (puede ser de 433, 868 o 915 MHz).

Todas las características que nos ofrece esta placa han sido de gran utilidad para desarrollar el sistema de forma simple para el usuario (radio LoRa, zona WiFi, ...), de forma autónoma (servidor web, almacenamiento interno propio, ...) y con un consumo de energía bajo.

5.2 Software

El software que se ha escogido para desarrollar la solución es variado.

El principal sería el IDE de Arduino, con el entorno para la placa ESP32 añadido (ya que no viene por defecto). De esta forma, el ESP32 se programaría de forma muy similar a una placa Arduino (con los típicos apartados de setup y loop). Para usar las características propias del ESP32 es necesario importar una serie de librerías. En el caso del programa desarrollado, estas han sido las librerías utilizadas junto con los enlaces a sus respectivos repositorios:

- **WiFi.h**: Usada principalmente para crear la zona WiFi.

https://github.com/espressif/arduino-esp32/blob/master/libraries/WiFi/src/WiFi.h

- **ESPAsyncWebServer.h**: Usada para implementar el servidor web.

https://github.com/me-no-dev/ESPAsyncWebServer

- **WebSocketsServer.h**: Utilizada para implementar la comunicación entre navegador y ESP32 vía WebSockets.

https://github.com/Links2004/arduinoWebSockets

- **Wire.h**: Necesaria para el uso de LoRa.

https://github.com/esp8266/Arduino/blob/master/libraries/Wire/Wire.h

- **SPI.h**: Necesaria para el uso de LoRa.

https://www.arduino.cc/en/Reference/SPI

- **LoRa.h**: Usada como interfaz para utilizar el módulo de LoRa.

https://github.com/sandeepmistry/arduino-LoRa

- **FS.h**: Necesaria para el uso del sistema de ficheros internos del ESP32.

https://github.com/esp8266/Arduino/blob/master/cores/esp8266/FS.h

- **SPIFFS.h**: Utilizada para gestionar el sistema de ficheros interno del ESP32.

https://github.com/espressif/arduino-esp32/blob/master/libraries/SPIFFS/src/SPIFFS.h

- **sqlite3.h**: Implementación de SQLite 3 para ESP32. Se ha utilizado como base de datos.

https://github.com/LuaDist/libsqlite3

- **mbedtls/md.h**: Utilizada para implementar la función de hash SHA-256. Funcionalidad embebida en el ESP32 por defecto.

https://github.com/ARMmbed/mbedtls/blob/master/include/mbedtls/md.h

- **hwcrypto/aes.h**: Utilizada para implementar la función de encriptación/desencriptación AES-256 CBC. Funcionalidad embebida en el ESP32 por defecto.

https://github.com/espressif/arduino-esp32/blob/master/tools/sdk/include/esp32/hwcrypto/aes.h

El resto de software (la interfaz gráfica) ha sido programado en HTML, CSS y Javascript. No requieren un entorno específico para su programación (basta un editor de texto). Su ejecución se ha realizado en navegadores, principalmente Firefox, aunque se han hecho pruebas en Chrome y Safari.

6. Sistema

6.1 Descripción general

El sistema LoRaChat al completo, en el que se incluyen los subsistemas de usuario y de red, estaría descrito de la siguiente manera:

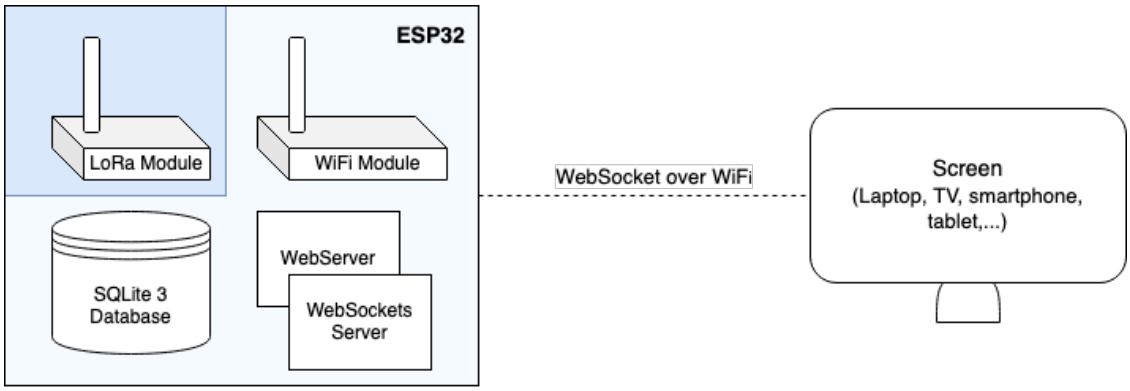


Fig. 11 Sistema LoRaChat

Por un lado, tendríamos, como se ha dicho, el subsistema visible para el cliente. Es decir, aquella parte del sistema LoRaChat con la que el cliente interactúa para hacer uso del servicio. Este, en principio, no supondría más que la pantalla del terminal en la que el usuario se conecta al ESP32. Aunque, en realidad, habría que tener en cuenta que para su funcionamiento son necesarios el servidor web, el servidor de WebSockets, la base de datos y el módulo WiFi. Todo ello es realizado por un único ESP32 que hará de puente entre un cliente y la red (el resto de los clientes o nodos).

Por otro lado, estaría el subsistema que se encarga de enviar, recibir y enrutar los paquetes de LoRaChat vía radio (LoRa) y que no es visible para el usuario.

6.2 Client side

El subsistema del cliente es el encargado de permitir al usuario interactuar con el sistema. Para ello, interpone una interfaz gráfica entre el usuario y la red.

La interfaz se sirve por medio de un servidor web ejecutado por el ESP32. Al principio se pensó en crear interfaces gráficas monolíticas como ejecutables de sistema operativo. O sea, se pensó en desarrollar programas ejecutables en el ordenador del usuario o aplicaciones que se ejecutarían en el smartphone. Sin embargo, esto suponía un gran despliegue en el desarrollo de la aplicación, pues había que decidir entre desarrollar para una sola plataforma o sistema operativo (lo que restringiría bastante su uso), o

desarrollar una aplicación por cada sistema operativo de entre un conjunto razonable en cuanto a estadísticas de uso (Windows, OSX, Linux, Android, iOS).

Visto el problema que suponía esto, se pensó en una primera alternativa: desarrollarlo en un entorno cross platform. El primero en el que se pensó fue Python, pues éste es capaz de ejecutarse en Windows, OSX y Linux sin problemas y sin apenas tener que hacer cambios entre plataformas. Pero esta alternativa dejaba fuera a los sistemas operativos de smartphone. Entonces, se buscó otra alternativa. Ésta fue ElectronJS.

Efectivamente, ElectronJS permitía ejecutarse en diferentes plataformas con cambios mínimos o nulos entre ellas. También existía la posibilidad de integrarse en entorno de smartphone, aunque no de manera tan trivial como se pensaba. Una aplicación desarrollada en ElectronJS se basa, principalmente, en HTML, CSS y Javascript. Es, a fin y al cabo, una aplicación web, pero hecha local y compactada en forma de aplicación de escritorio. Debido a esto, aumentaban las posibilidades de programar una interfaz gráfica atractiva y moderna, con una gran variedad de diseños, a diferencia de las librerías de interfaz gráfica de Python, como Tkinter o PyQT.

Cuando la opción de desarrollar la GUI en ElectronJS parecía la definitiva, surgió una idea que soluciono todas las problemáticas que suponían las dos alternativas anteriores: programar una página web y servirla en un servidor embebido en el ESP32.

Esto supuso una serie de ventajas. Las primeras eran que ya no había que preocuparse por el entorno de desarrollo, la migración entre plataformas, la adaptación a sistemas operativos de smartphone, etc. Ya que simplemente se trataba de diseñar una página web con HTML y CSS, donde todo el controlador se ejecutase en Javascript. La segunda ventaja residía en el hecho de que el usuario ya no tenía la necesidad de instalarse una aplicación en su computadora o en su smartphone: ahora bastaba con cualquier dispositivo con un navegador y la capacidad de conectarse a una red WiFi. De esta forma, la interfaz se convertía, con todas sus reservas, en una interfaz universal: ejecutable en computadora, en smartphone, en tablet, en smart TV, etc.

A partir de este momento y definitivamente, el desarrollo de la interfaz gráfica se realizó con esta idea en mente. Dos eran los componentes por desarrollar: la página web y el servidor web.

La página web comenzó a diseñarse primero de forma gráfica. Se trataba de diseñar una interfaz de chat agradable a la vista, fácil de usar y acorde una línea minimalista. Esta se basó en las interfaces de aplicaciones de chat como WhatsApp, Signal y otras.

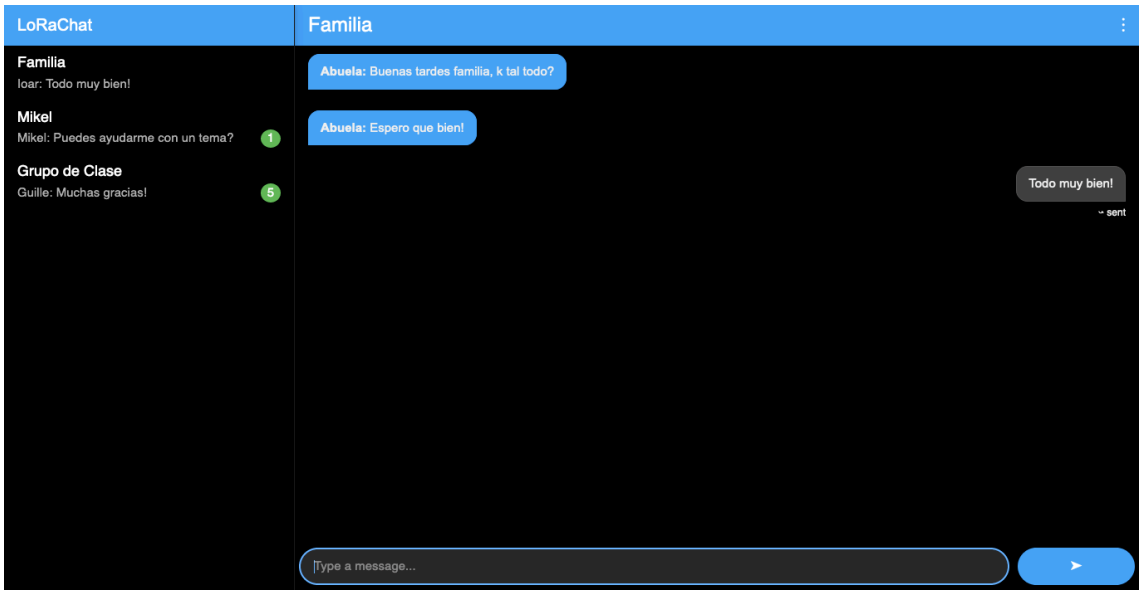


Fig. 12 Pantalla de chat en la GUI de LoraChat

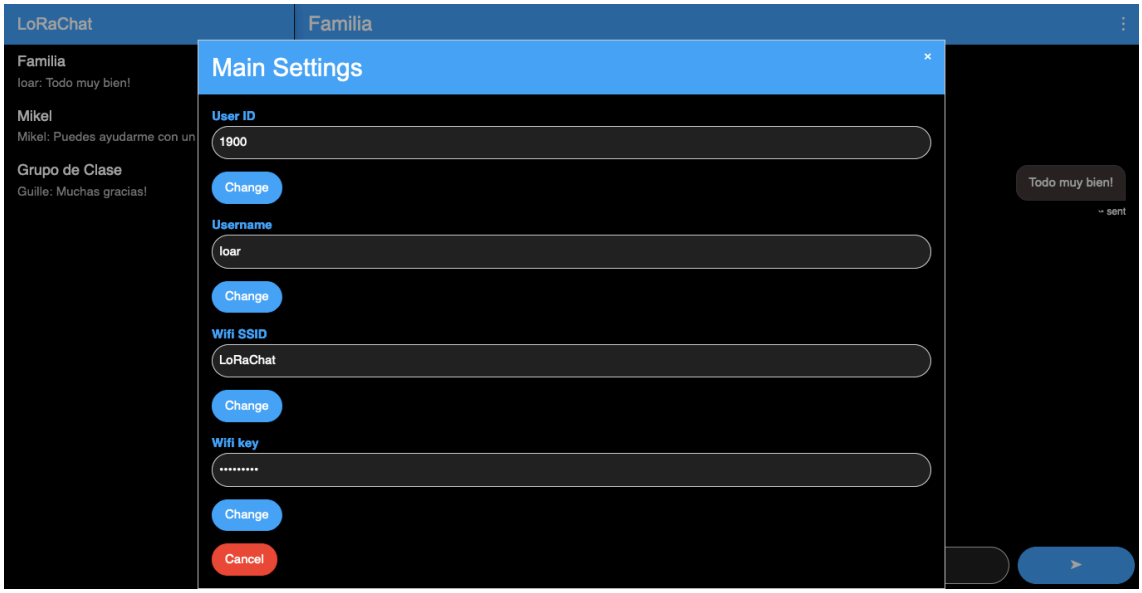


Fig. 13 Ajustes principales en la GUI de LoRaChat

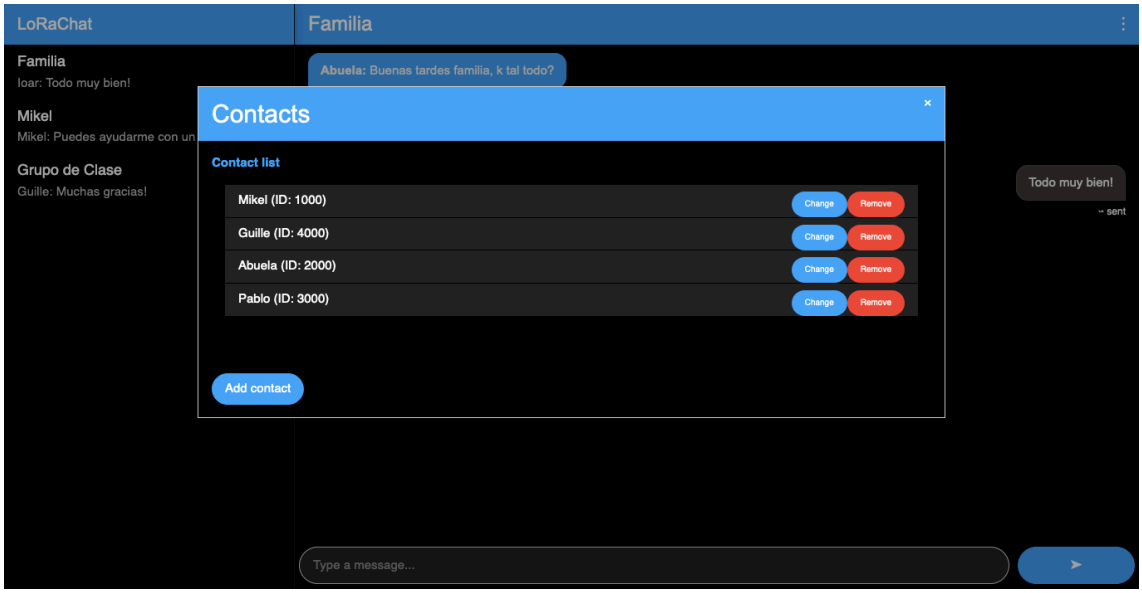


Fig. 14 Ajustes de contactos en la GUI de LoRaChat

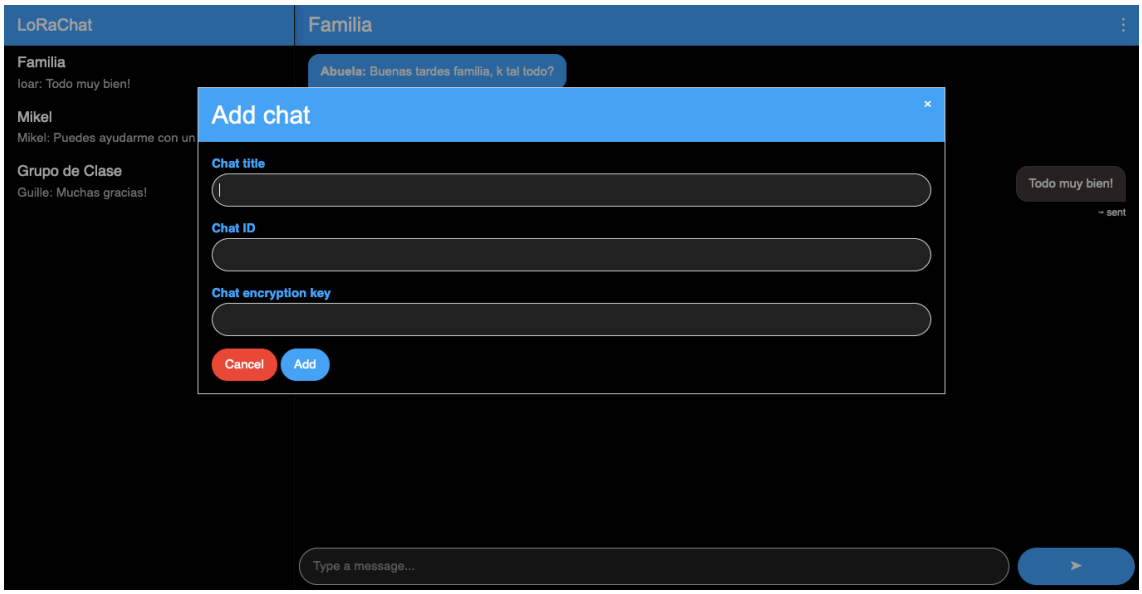


Fig. 15 Añadir chat en la GUI de LoRaChat

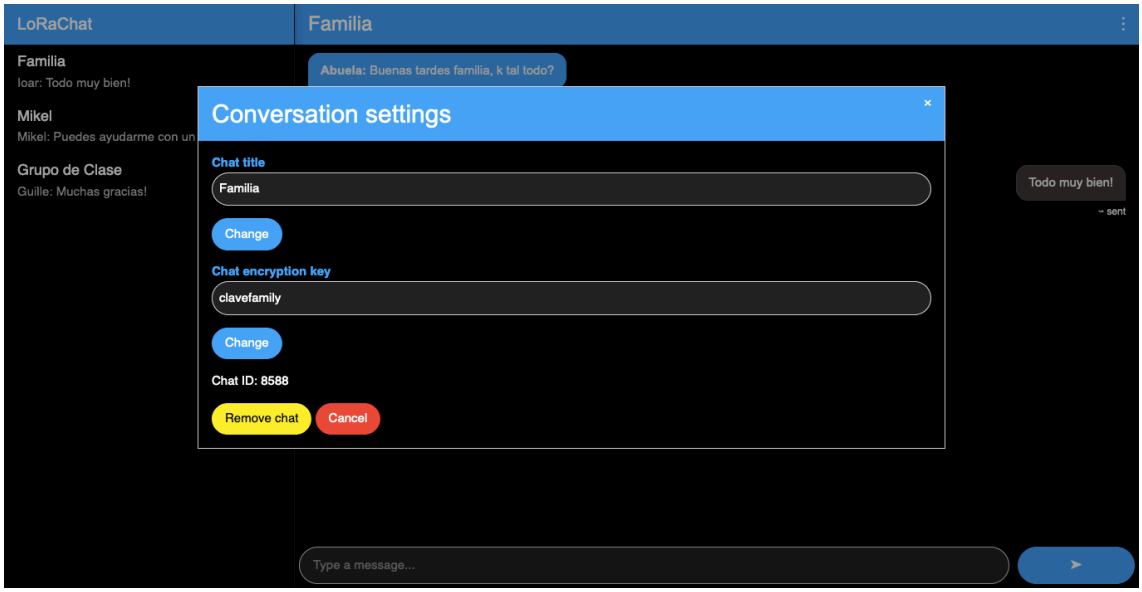


Fig. 16 Ajustes de conversación en la GUI de LoRaChat

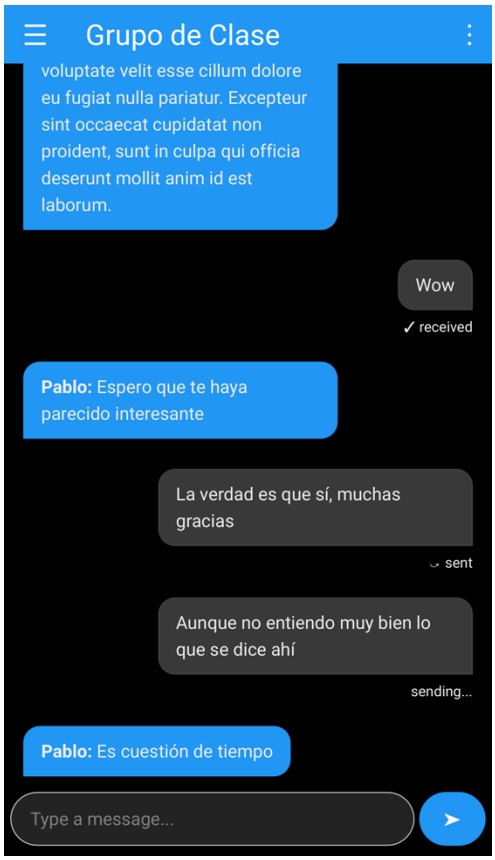


Fig. 17 Pantalla de chat en la GUI de LoRaChat en smartphone

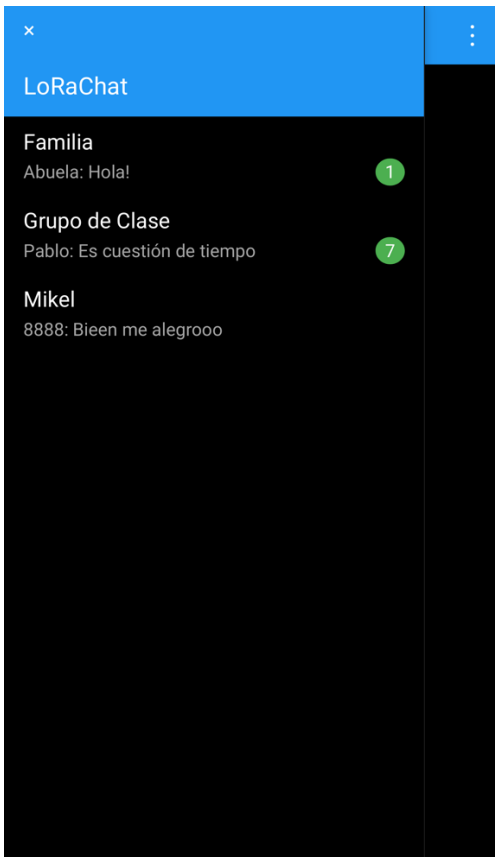


Fig. 18 Barra lateral con lo chats en la GUI de LoRaChat en smartphone

Por otro lado, se desarrolló el servidor web asíncrono que se ejecutaría en el ESP32. Para ello también era necesario poner en marcha una red para poder conectarse al servidor, optándose por una zona WiFi local en el ESP32. De este modo, el usuario se conectaría con cualquier dispositivo a la red WiFi creada por el ESP32 y en el navegador se le serviría la interfaz gráfica, de la que haría uso para chatear con otros usuarios de la red LoRaChat.

Para conectar la página web con el servidor de tal modo que una vez abierta en el navegador del cliente se pudiese comunicar con el ESP32 de forma fluida, se optó por abrir un WebSocket, ya que el ESP32 tiene una buena y sencilla librería para ello y Javascript, por su parte, trae de forma nativa la implementación del protocolo.

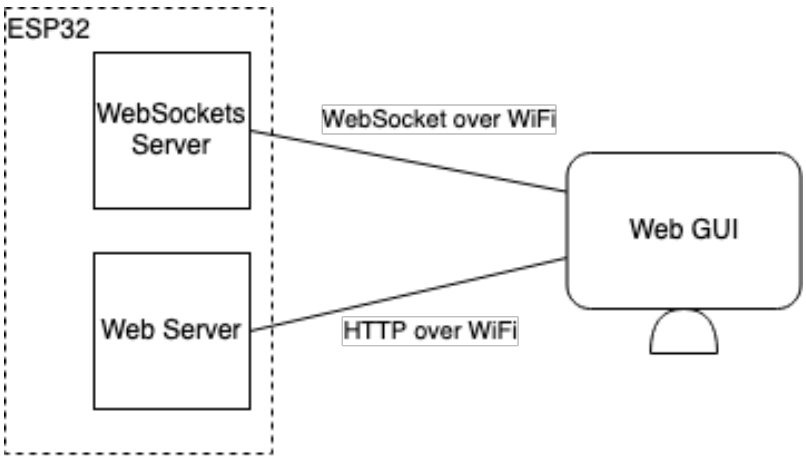


Fig. 19 Conexión entre pantalla y ESP32

Así, el proceso comenzaría con la creación de la zona WiFi y la puesta en marcha del servidor web, cuando el ESP32 se enciende. Después, el usuario se conectaría a la zona WiFi y abriría en el navegador la página web. En ese momento, se abre un WebSocket entre navegador y ESP32, donde éste último es el servidor.

A partir de aquí, toda interacción del usuario con la interfaz gráfica se procesaría mediante el código de Javascript ejecutado en el navegador. Las funciones de Javascript en la GUI son gestionar los cambios visuales en la interfaz gráfica cuando son requeridos (abrir y cerrar ventanas, añadir mensajes, crear notificaciones, etc.) y gestionar el intercambio de paquetes de información entre la página web y el ESP32 (envío y recepción de mensajes LoRaChat, envío y recepción de información de la base de datos, etc.).

Cuando se recibe un mensaje por radio, el ESP32 lo procesa y, si es para el usuario y éste está conectado la interfaz, lo envía por WebSocket. Si no, lo guarda en un buffer hasta que el usuario se conecte a la GUI, de forma que el dispositivo pueda funcionar en modo standalone. Y, al revés, cuando el usuario envía un mensaje a través de la interfaz gráfica, el código Javascript procesa los datos, realiza los cambios necesarios en la interfaz y envía la información del mensaje por WebSocket, para que sea recogido por el ESP32 y éste lo envíe por LoRa.

Con la base de datos se realizan acciones parecidas. Cuando el usuario se conecta a la interfaz gráfica, el ESP32 envía toda la información guardada en la base de datos para ser mostrada en la GUI. Y, cuando el usuario realiza algún cambio crucial (véase el apartado de la base de datos), el código de Javascript envía por WebSocket esa información para que el ESP32 la almacene.

Como puede verse, los únicos dispositivos que son necesarios para usar LoRaChat son el ESP32 encargado de servir la GUI y enlazar ésta con la comunicación por radio, y un navegador ejecutado en cualquier dispositivo conectable a la red WiFi.

6.3 Red LoRaChat y radio

Para las acciones relacionadas con el envío, recepción y procesamiento (encriptación/desencriptación, parsing, ensamblaje, etc.) de paquetes, el encargado es el ESP32. El dispositivo se mantiene todo el rato que está encendido a la escucha en el módulo LoRa, para recibir paquetes, a la par que, si el usuario está conectado, escucha por WebSocket para recibir mensajes y enviarlos por radio. También se ocupa del enrutamiento de paquetes de la red LoRaChat.

El proceso para enviar paquetes es muy sencillo. Cuando el ESP32 recibe un paquete por WebSocket con el flag de tipo en el que se indica que es un mensaje LoRaChat, entonces recoge los datos del paquete (ID de talkgroup, autor, ID de mensaje y texto) y comienza el proceso de envío. Este proceso pasa por la encriptación, creación de hashes y ensamblaje del paquete (explicado en el apartado 7.2). Una vez montado el paquete, si el módulo está disponible para enviar, el paquete se envía por radio.

En la recepción se realizaría el mismo proceso, pero a la inversa, como se detalla en el apartado 7.3. De forma resumida, se recibe un paquete y se realizan las acciones de procesamiento, parsing, desencriptación y, finalmente, almacenamiento en buffer o envío por WebSocket, según la disponibilidad del usuario.

En este proceso de recepción, también se realiza el proceso de routing, explicado en el apartado 7.3. Se comprobaría si el paquete recibido existe en la tabla de paquetes recientes y, en caso de no existir, se reenvía por radio.

7. Protocolo de comunicación

El protocolo de comunicación puede dividirse en dos partes principales: la secuencia de intercambio de paquetes y la composición de los paquetes. Dentro de esta última, es necesario describir la forma en la que se procesa un paquete.

7.1 Paquetes

Para un uso básico del sistema LoRaChat, se han diseñado dos paquetes esenciales. Por un lado, está el mensaje de texto, utilizado para enviar los mensajes de texto de usuario a través de la red. Por otro lado, está el ACK, cuya función es notificar a la otra parte de la recepción de un paquete.

Ambos paquetes disponen de tres campos generales, de los cuales solo cambia el último, en función del tipo de paquete. Estos campos son el ID (hash) de paquete, el ID (hash) de talkgroup y el payload.

El ID (hash) de paquete identifica de forma única a cada paquete, en base a su contenido. Es un hash del payload del paquete sin encriptar. Aparte de la identificación, también sirve para comprobar la integridad del paquete.

El ID (hash) de talkgroup informa sobre el talkgroup al que pertenece el paquete. La razón de ello reside en que es necesario que los nodos sepan si el paquete va dirigido a un talkgroup al que pertenezcan o no, para pasar a las siguientes fases de procesamiento. También sirve para comprobar la integridad del paquete, comparándose con el ID de talkgroup que aparece en el payload.

El payload contiene la información de chat: el ID de talkgroup, el tipo de mensaje (de texto o ACK), el autor, el ID de mensaje y, si es de tipo mensaje de texto, el texto.

Los valores que puede tomar el flag del tipo de paquete son:

#define LORA_MSG 0

Para un mensaje de texto

#define LORA_ACK 1

Para un ACK.

Paquete de texto:

Un paquete de mensaje de texto tendría la siguiente forma:

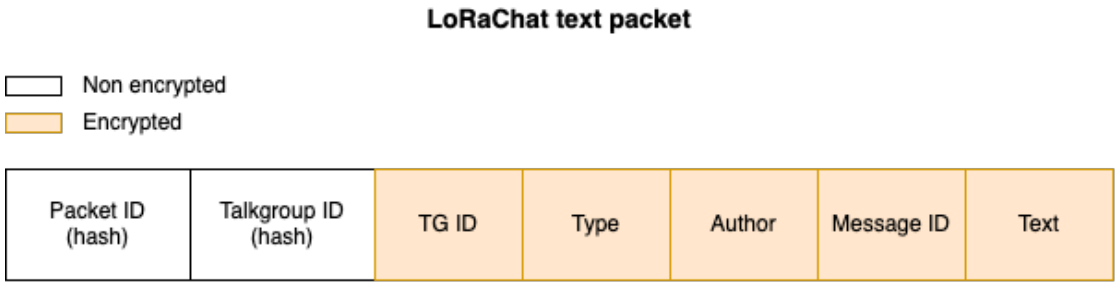


Fig. 20 Paquete de texto LoRachat

Como puede verse, el paquete puede dividirse principalmente en id del paquete (hash), id del talkgroup (hash) y el payload encriptado. Dentro del payload encriptado van encapsulados el id del talkgroup (sin hash), el tipo de mensaje (en este caso, de texto, lo que equivale a 0), el autor del mensaje (identificador), el id del mensaje y el texto plano del mensaje.

Cabe señalar que el id del mensaje sólo tiene sentido cuando forma la terna talkgroup-autor-id, pues el id se asigna en base a un contador que es único por cada autor en un talkgroup determinado. Por ejemplo, si un usuario envía su primer mensaje en un determinado talkgroup, el id de mensaje simple será 0, igual que el de cualquier otro usuario que escriba su primer mensaje en ese talkgroup. Pero lo que lo hará único será la terna señalada, mediante la cual, en el hipotético caso de que el talkgroup fuese el 9000 y el id del autor fuese 5934, se definiría como 9000-5934-0.

La función del id de paquete (hash), es la de identificar cuasi únicamente (pues hay que recordar que el hash es recortado, aunque es muy improbable que coincidan en el tiempo dos paquetes con el mismo hash) el paquete, a la par que sirve para comprobar la integridad del paquete encriptado.

El ID de talkgroup hasheado, por su parte, tiene dos objetivos, aparte de la comprobación de integridad. Uno es el de facilitar al sistema encontrar la clave para desencriptar el paquete (en el caso de que sea un receptor legítimo), a la par que, y este es el segundo objetivo, se oculta el id original del talkgroup, en aras de mantener en secreto los IDs de talkgroup ante la recepción del paquete por alguien que no sea el legítimo destinatario.

Paquete ACK

Un paquete ACK se describe de la siguiente manera:

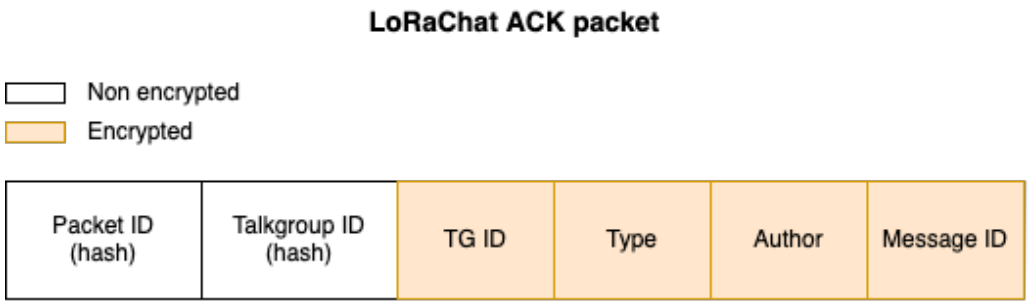


Fig. 21 Paquete ACK LoRaChat

Como puede verse, tiene casi todos los mismos campos que en un paquete de texto, excepto el campo en el que se incluye el mensaje en texto plano, porque no es objeto del ACK enviar texto, sino simplemente notificar de que el texto ha sido recibido. Para ello, basta simplemente con enviar la información del payload del mensaje de texto cuya recepción se desea notificar, cambiando el tipo de mensaje por el indicador de ACK, el cual corresponde a 1.

Es importante tener en cuenta que al cambiar el payload, también cambia el id del paquete (hash). El resto del mensaje es invariable.

Cada campo tiene las mismas funciones que las de un mensaje de texto, explicadas en el apartado anterior.

7.2 Secuencia de envío de paquetes

La secuencia en modalidad simplex, es decir, en el que únicamente dos nodos establecen una comunicación directa, vendría descrita por la siguiente imagen:

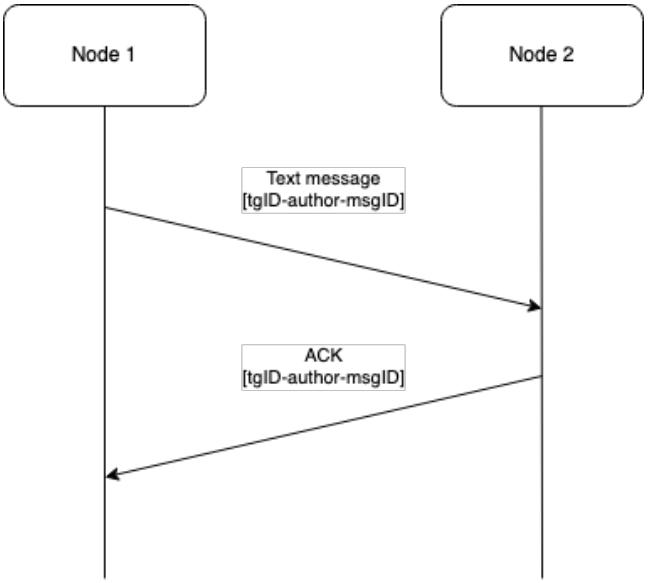


Fig. 22 Secuencia de envío de mensaje en simplex en LoRaChat

Un usuario envía un mensaje haciendo inundación (porque no hay enlaces) y el otro usuario lo recibirá (o no) y, en el caso de que disponga de la clave del grupo de chat, procesará el mensaje y contestará con un ACK, de modo que el primer usuario sepa que el segundo ha recibido el mensaje.

Existe la posibilidad de operar únicamente en simplex, estableciendo el valor del flag ROUTING a 0. Sin embargo, la modalidad simplex no es la forma original de operar de LoRaChat, siendo ésta la modalidad en malla (el flag ROUTING con valor 1).

Para describir la secuencia del comportamiento del envío de paquetes en la red mallada, se van a utilizar únicamente dos nodos, con el fin de simplificar la explicación. En cualquier caso, en capítulos posteriores se verá cómo al usar más de dos nodos la red empieza a dar problemas. También se va a introducir una tercera entidad que representa el resto de los nodos de la malla, para así poder explicar a dónde van los mensajes (aunque de la malla no se esperará respuesta, cuando en realidad puede que la haya). El nodo 1 alcanza al nodo 2, pero no al resto de la malla. El nodo 2, por su parte, alcanza al nodo 1 y a uno o varios nodos del resto de la malla:

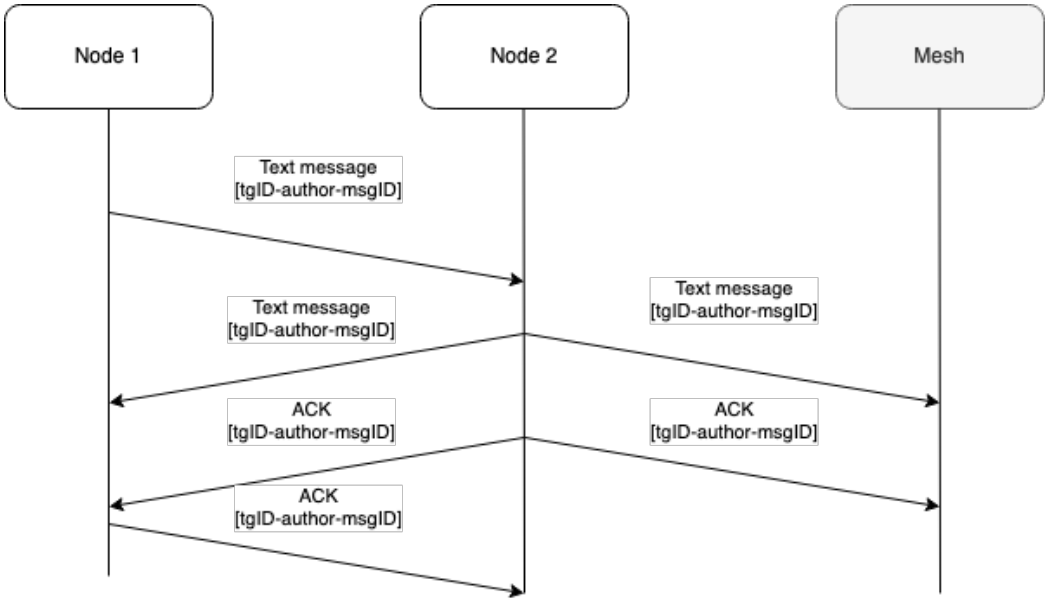


Fig. 23 Secuencia de envío de paquetes en malla LoRaChat

Un usuario envía un mensaje haciendo inundación y este mensaje lo recibirán uno o varios (o ninguno) usuarios o nodos. Cada nodo comprobará si el paquete fue recibido recientemente. Solo en caso de que no haya sido recibido recientemente se volverá a reenviar por la red haciendo inundación. Después, cada nodo enviará el ACK correspondiente si procede.

Este proceso es realizado por cada nodo al que le llegue el mensaje del primer usuario.

Por último, faltaría por explicar el proceso de encriptación y ensamblaje de paquetes para su envío. Este proceso solo se realiza en el nodo que crea un determinado mensaje. Esto es así porque los paquetes que se reciben por radio y que habría que enrutar, no es necesario parsearlos ni ensamblarlos. Basta con reenviar el paquete tal cual ha sido recibido, en caso de que proceda su reenvío.

El proceso de ensamblaje y encriptación para un mensaje de texto es el siguiente (para un mensaje ACK el proceso es el mismo, pero sin el campo de texto):

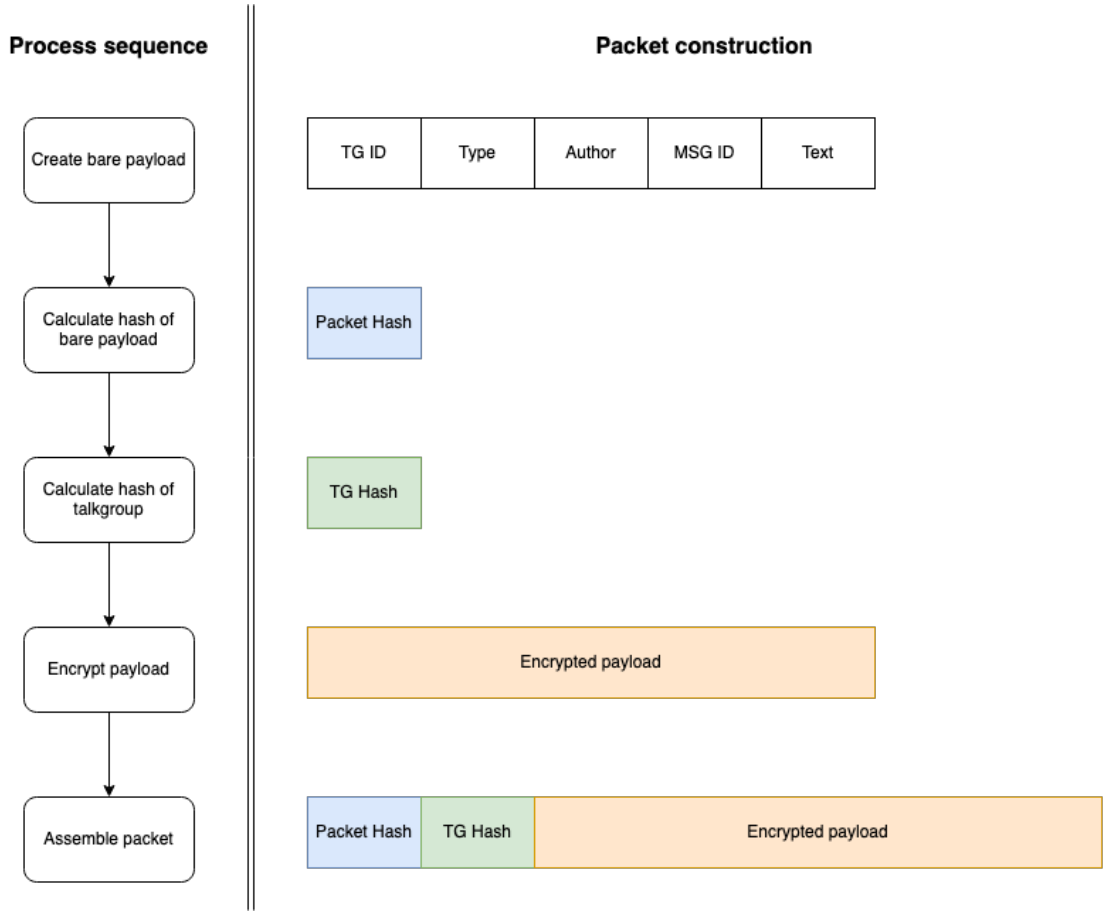


Fig. 24 Proceso de ensamblaje de paquete LoRaChat

Primero se crea el payload sin encriptar, donde se incluyen los campos ID de talkgroup, tipo, autor, ID de mensaje y, si es un paquete de mensaje de texto (y, por tanto, no es un ACK), se incluye el mensaje escrito por el usuario. Una vez creado este payload, se calcula su hash para la creación del ID y, también, el hash del talkgroup con el mismo propósito, pero aplicado a este campo. Después, el payload se encripta. Llegados a este punto todos los campos del paquete estarían creados, por lo que se añaden a un paquete LoRa, divididos por separadores: primero el ID (hash) del paquete, luego el ID (hash) del talkgroup y, finalmente, el payload encriptado. Acto seguido, si el módulo LoRa está libre, el paquete es enviado.

7.3 Secuencia procesamiento de paquetes recibidos

Cuando se recibe un paquete, se efectúa el siguiente proceso:

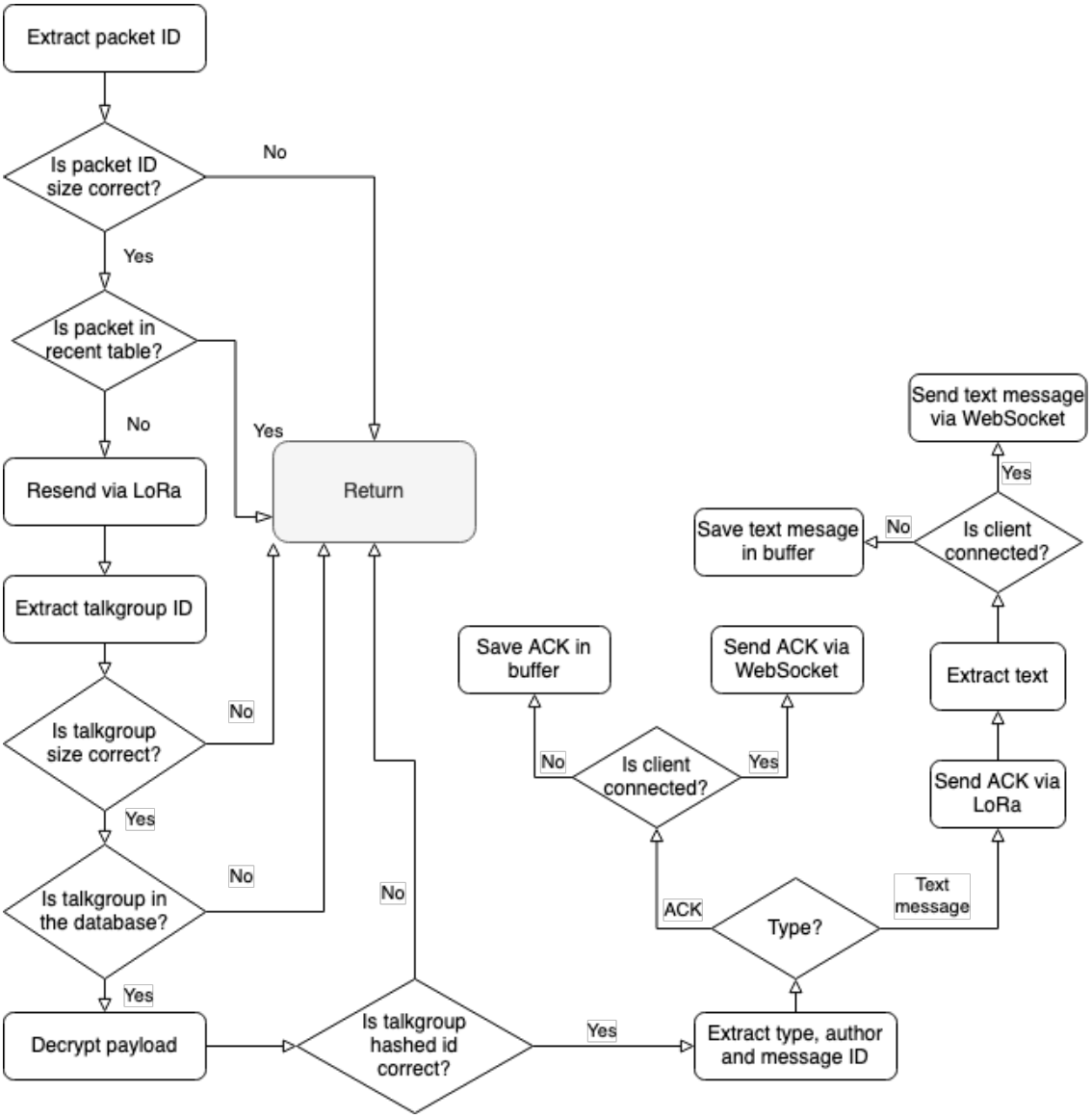


Fig. 25 Secuencia de procesamiento de paquetes LoRaChat

Lo primero es extraer el ID (hash) del paquete, comprobar que el tamaño del ID es correcto y comprobar si el paquete está en la tabla de recientes. En caso de que aparezca en la tabla de recientes, significa que el paquete fue recientemente recibido y que ya fue procesado, por lo que se procede a desecharlo. Si el paquete no está en recientes, entonces se continua con el proceso.

A continuación, si el modo routing está activado, el paquete se reenviará por radio.

Después, se guarda en la tabla de recientes, para tener constancia en el futuro de que el paquete ha sido procesado.

Lo siguiente sería extraer el ID (hash) del talkgroup y comprobar que el tamaño es correcto. Si lo es, se pasa a comprobar si existe en la tabla de talkgroups de usuario. Es decir, se comprueba si el nodo en el que se está realizando el proceso es un receptor legítimo. Si no existe, se desecha. En caso contrario, se consulta la clave de desencriptado del paquete asociada al talkgroup.

Ahora es cuando se procesa el payload. En primer lugar, se desencripta su contenido con la clave del talkgroup. En segundo lugar, se comprueba la integridad comparando al ID del paquete (hash) con un hash del payload desencriptado procesado al momento. Si coinciden, el paquete es correcto y se continua. Si no, se desecha el paquete.

En ese payload se realiza una última comprobación de integridad, en la cual se obtiene el hash del ID de talkgroup que viene dentro del payload y se compara con el ID de talkgroup que venía en la cabecera del paquete.

Una vez aprobada esta comprobación, el paquete es totalmente correcto y se procede a extraer los primeros datos: tipo de mensaje, autor e ID del mensaje. Si el mensaje es de tipo texto (0), lo primero es enviar por radio el ACK correspondiente al mensaje y, entonces, se continúa extrayendo el campo de texto que contiene el mensaje en sí y se realizan las acciones para notificar al usuario y mostrar el mensaje. En caso de ser un mensaje ACK (1), lo único que se hace es realizar las operaciones para notificar al usuario del acuse de recibo.

Hay que tener en cuenta que, en esta última fase del procesamiento, solo se realizan las acciones de notificación al usuario en caso de que el usuario haya abierto la interfaz gráfica. Si no es el caso, los mensajes se almacenan en un buffer para ser enviado más tarde, cuando el usuario se conecte a la interfaz web.

7.4 Paquetes de WebSocket

El objetivo de los paquetes que se intercambian por medio del WebSocket son los de gestionar los eventos que ocurren tanto en la interfaz gráfica como en el ESP32 y que requieren la interacción entre ambos elementos.

El formato de los paquetes (a nivel de aplicación) que se envían difiere radicalmente según el origen del envío. Si es enviado desde el ESP32 a la página web, el formato de los paquetes es JSON. Si, en cambio, el paquete se envía desde el navegador al ESP32, el paquete tiene un formato de desarrollo propio. La razón de esta decisión está en la capacidad que cada entorno tiene para procesar cada tipo de paquete. En Javascript, la estructura de datos por excelencia es JSON. Javascript gestiona de forma muy eficiente y hasta trivial este tipo de formato. Sin embargo, en el caso del ESP32, si bien existe alguna librería de JSON para Arduino, este tratamiento no es tan efectivo. Y en tanto que los paquetes no son de una complejidad tal como para implementar el formato JSON, se ha optado por una opción más simple como es el uso de separadores para delimitar campos.

Un paquete (a nivel de aplicación) enviado por WebSocket desde el ESP32 al navegador web tiene esta forma genérica:

```
{
        "type": FLAG_TIPO,
        "dato1": dato1,
        "dato2": dato2,
        ...
        "datoN": datoN
}
```

Donde el tipo (type) sería uno de los valores que se explicarán más abajo y, el resto de los campos (dato1, dato2, ..., datoN) los datos necesarios para realizar la operación.

Un paquete (a nivel de aplicación) enviado por WebSocket desde el navegador al ESP32 tiene esta forma genérica:

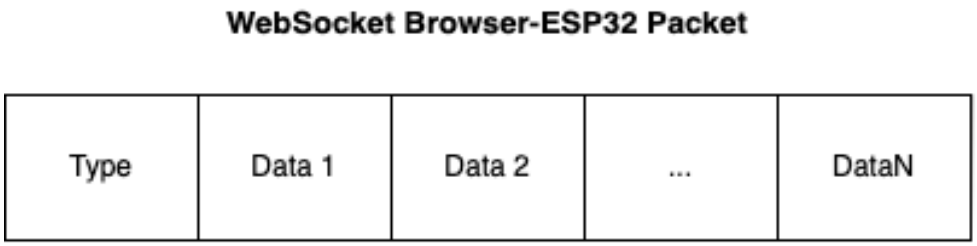


Fig. 26 Paquete intercambiando entre ESP32 y navegador web vía WebSockets

Donde el tipo (type) sería uno de los valores que se explicarán a continuación y, el resto de los campos (dato1, dato2, ..., datoN) los datos necesarios para realizar la operación.

Los valores que puede tomar el flag de tipo (type) de un paquete son estos:

#define WS_ERROR -1

Utilizado cuando hay algún error en el procesamiento de alguna petición. Le acompaña un único dato que contiene el valor del flag del tipo de mensaje que ha dado lugar al error. Este tipo de mensaje puede ser enviado en ambas direcciones.

#define WS_SEND_MSG 0

Utilizado cuando se quiere enviar un mensaje. Los datos que acompañan a este tipo son: ID del talkgroup, ID del mensaje, ID del autor y el texto del mensaje. Este tipo de mensaje solo es enviado desde el navegador al ESP32.

#define WS_RECV_MSG 1

Utilizado cuando se ha recibido un mensaje. Es acompañado por: ID del talkgroup, ID del mensaje, ID del autor y el texto del mensaje. Este tipo de mensaje solo es enviado desde el ESP32 al navegador.

#define WS_RECV_ACK 2

Utilizado cuando se ha recibido un ACK. Es acompañado por: ID del talkgroup, ID del mensaje e ID del autor. Este tipo de mensaje solo es enviado desde el ESP32 al navegador.

#define WS_MSG_SENT 3

Utilizado para notificar al navegador de que un mensaje ha sido correctamente enviado por radio. Es acompañado por: ID del talkgroup, ID del mensaje e ID del autor. Este tipo de mensaje solo es enviado desde el ESP32 al navegador.

#define WS_DB_REQ 4

Utilizado para hacer una petición de la base de datos. No es acompañado por ningún dato más. Este tipo de mensaje solo es enviado desde el navegador al ESP32. Una vez enviado, la respuesta esperada es la base de datos almacenada en el ESP32 al completo.

#define WS_DB_SEND 5

Utilizado para enviar la base de datos. El único dato que lo acompaña es la base de datos completa en formato JSON. Este tipo de mensaje puede ser enviado en ambas direcciones.

#define WS_DB_RECV 6

Utilizado para confirmar la recepción de la base de datos. No es acompañado por ningún dato más. Este tipo de mensaje solo es enviado desde el ESP32 al navegador.

#define WS_ADD_CHAT 7

Utilizado para añadir un talkgroup en la base de datos. Le acompañan el ID del talkgroup y su clave de encriptación. Este tipo de mensaje solo es enviado desde el navegador al ESP32.

#define WS_DEL_CHAT 8

Utilizado para eliminar un talkgroup en la base de datos. Le acompaña el ID del talkgroup a eliminar. Este tipo de mensaje solo es enviado desde el navegador al ESP32.

#define WS_SET_CHAT_KEY 9

Utilizado para modificar la clave de un talkgroup en la base de datos. Le acompaña el ID del talkgroup y la nueva clave de encriptación. Este tipo de mensaje solo es enviado desde el navegador al ESP32.

#define WS_SET_WIFI_SSID 10

Utilizado para modificar el SSID de la zona WiFi del ESP32 en la base de datos. Le acompaña el nuevo SSID de la zona WiFi. Este tipo de mensaje solo es enviado desde el navegador al ESP32.

#define WS_SET_WIFI_KEY 11

Utilizado para modificar la clave de la zona WiFi del ESP32 en la base de datos. Le acompaña la nueva clave de la zona WiFi. Este tipo de mensaje solo es enviado desde el navegador al ESP32.

#define WS_BUFFERED_MSG 12

Utilizado para solicitar los mensajes almacenados en el buffer. No le acompaña ningún dato. Este tipo de mensaje solo es enviado desde el navegador al ESP32. La respuesta esperada es la recepción de los mensajes almacenados en buffer, uno a uno, mediante paquetes de WebSocket con el flag de tipo con valor WS_RECV_MSG (1).

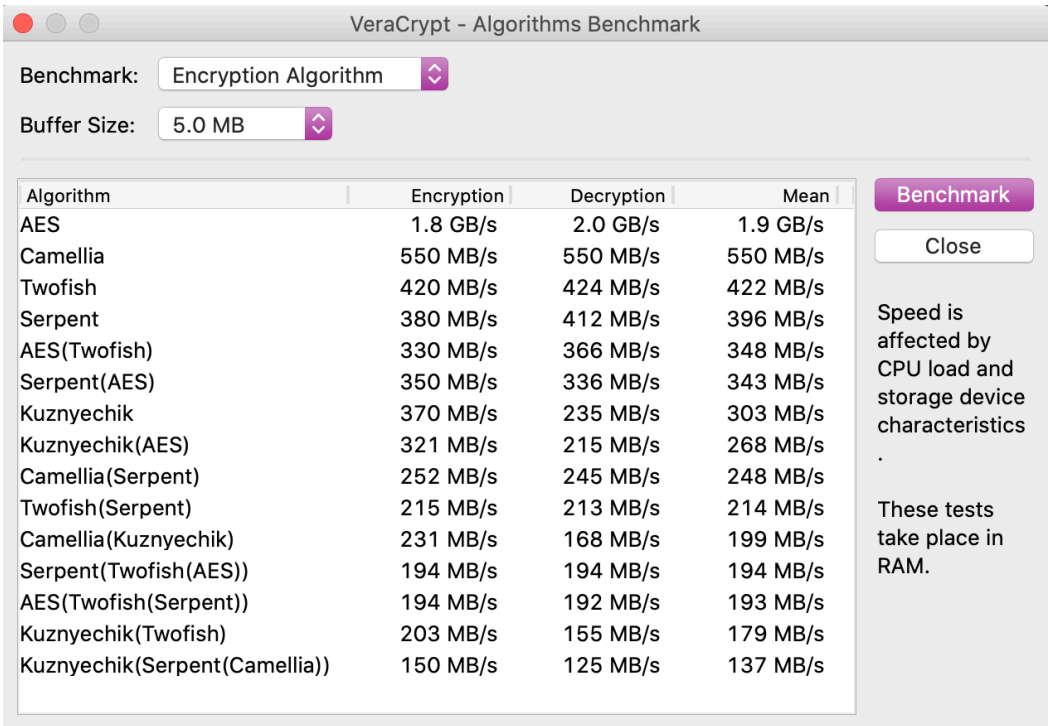
8. Seguridad y encriptación

En este apartado se pretende describir los métodos de encriptación que se han aplicado en los paquetes del protocolo, así como los cálculos de hash para generar IDs únicos y comprobar la integridad de los paquetes.

8.1 Encriptación de payload

Para encriptar el payload de los paquetes, se ha optado por utilizar AES-256 en modo Cipher block chaining (CBC).

Advanced Encryption Standard (AES) es un algoritmo de encriptación simétrico utilizado por entidades gubernamentales de peso como la National Security Agency estadounidense. Sus principales ventajas son su alto nivel de seguridad y su rapidez, dos factores de gran interés para este proyecto **[20]**. Por un lado, por la necesidad de ocultar el mensaje original de forma eficaz. Por otro lado, debido a la necesidad de un rápido procesamiento de mensajes, de modo que no se cree un cuello de botella en la red.



Algorithm	Encryption	Decryption	Mean
AES	1.8 GB/s	2.0 GB/s	1.9 GB/s
Camellia	550 MB/s	550 MB/s	550 MB/s
Twofish	420 MB/s	424 MB/s	422 MB/s
Serpent	380 MB/s	412 MB/s	396 MB/s
AES(Twofish)	330 MB/s	366 MB/s	348 MB/s
Serpent(AES)	350 MB/s	336 MB/s	343 MB/s
Kuznyechik	370 MB/s	235 MB/s	303 MB/s
Kuznyechik(AES)	321 MB/s	215 MB/s	268 MB/s
Camellia(Serpent)	252 MB/s	245 MB/s	248 MB/s
Twofish(Serpent)	215 MB/s	213 MB/s	214 MB/s
Camellia(Kuznyechik)	231 MB/s	168 MB/s	199 MB/s
Serpent(Twofish(AES))	194 MB/s	194 MB/s	194 MB/s
AES(Twofish(Serpent))	194 MB/s	192 MB/s	193 MB/s
Kuznyechik(Twofish)	203 MB/s	155 MB/s	179 MB/s
Kuznyechik(Serpent(Camellia))	150 MB/s	125 MB/s	137 MB/s

Fig. 27 Comparativa de algoritmos de encriptación en VeraCrypt

AES tiene varias implementaciones. De todas ellas se ha querido utilizar la modalidad Cipher block chaining (CBC), con el objetivo de ocultar posibles patrones en la encriptación de los paquetes.

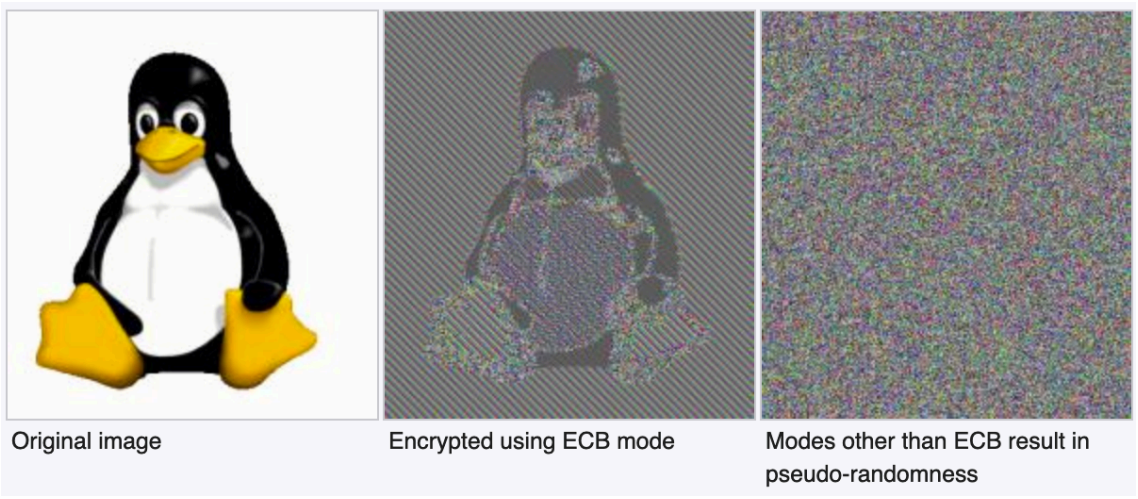
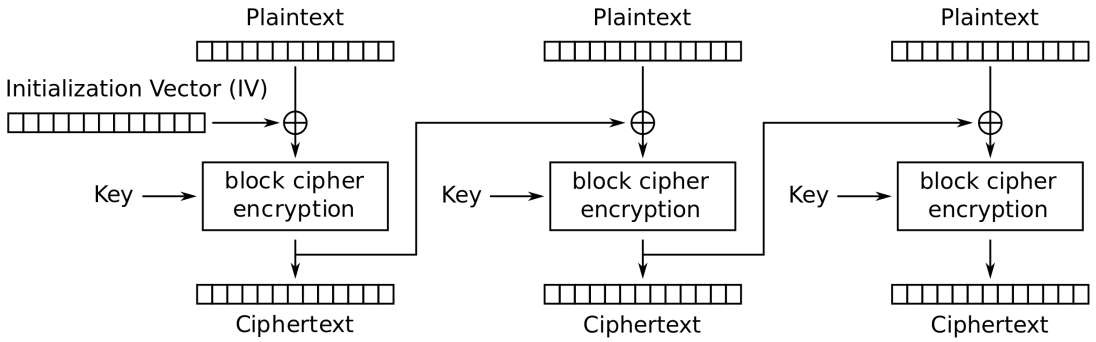


Fig. 28 Comparativa de encriptación de imagen con algoritmos ECB y CBC. Fuente: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

CBC es una modalidad que cifra bloque a bloque, utilizando la salida del proceso en el bloque anterior para encriptar el siguiente **[21]**. Este modo requiere de una secuencia binaria única, llamada Initialization Vector (IV), para producir una salida diferente para cada texto plano, incluso si ha sido encriptado con la misma clave. El tamaño de la clave que se ha escogido es de 256 bits, para obtener un nivel de seguridad alto (cuantos más bits de clave, más seguridad).



Cipher Block Chaining (CBC) mode encryption

Fig. 29 Diagrama de encriptación con CBC. Fuente: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:CBC_encryption.svg

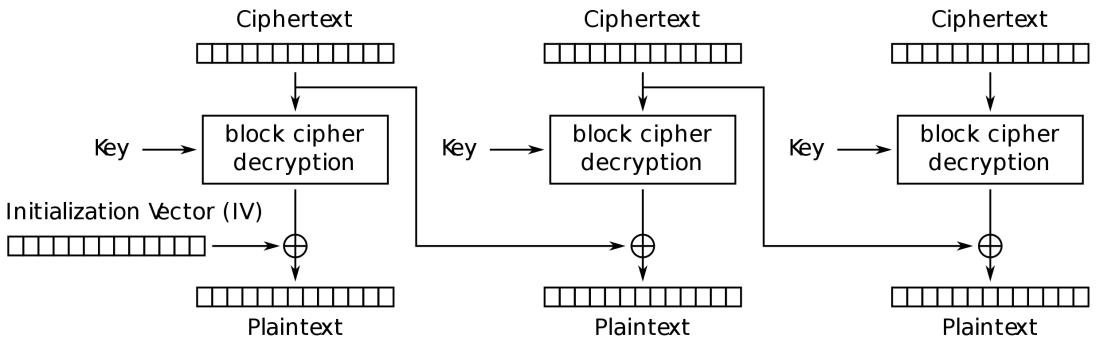


Fig. 30 Diagrama de desencriptación con CBC. Fuente: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:CBC_decryption.svg

Para implementar este algoritmo, se ha utilizado una librería que viene integrada por defecto en la API del ESP32, llamada "hwcrypto/aes.h". Como entrada se recibe el texto plano y la clave. Como la clave es la que haya elegido el usuario o el conjunto de usuarios de un talkgroup y, por comodidad para el usuario, éste puede escoger una clave alfanumérica cualquiera de longitud mayor que 8 caracteres, por lo que lo primero es transformar esa clave para que su tamaño sea de 256. Para ello se ha escogido el algoritmo de hash SHA-256, que se detallará en el siguiente apartado, pues para cualquier entrada siempre devuelve una salida de tamaño 256 bits. De este modo, ya tenemos una clave que cumple los requisitos de AES-256.

El IV, por su parte, se deriva a partir de la clave de cifrado y del hash del talkgroup. Así, pues, una misma contraseña producirá resultados diferentes para cada talkgroup diferente, haciendo más disperso el algoritmo y dificultando ataques de diccionario y fuerza bruta.

El algoritmo devolvería como salida el texto encriptado.

Para la desencriptación, el proceso es el mismo, pero a la inversa. La entrada es el texto encriptado y la clave, que debería ser la misma con la que se encriptó el contenido de la entrada. El resultado del algoritmo es el contenido desencriptado.

8.2 Algoritmo de hash

Como se explicaba anteriormente, el hash se ha utilizado, por un lado, para crear IDs únicos (casi únicos, más bien) para identificar paquetes o talkgroups y, por otro lado, para realizar comprobaciones de integridad. Estos son dos de los usos más comunes que se les dan a los algoritmos de hash.

El algoritmo concreto que se ha utilizado es el SHA-256 **[22]**. Este algoritmo pertenece al conjunto de funciones criptográficas de hash SHA-2 (Secure Hash Algorithm 2). Las razones de su elección residen en su considerable nivel de seguridad (aunque es cierto que existen vulnerabilidades ante ataques de extensión de longitud), por su rapidez frente a otros algoritmos de hash y, principalmente, porque provee un resultado de tamaño 256 bits, perfecto para su uso en el algoritmo AES-256 que se utiliza para la encriptación del payload.

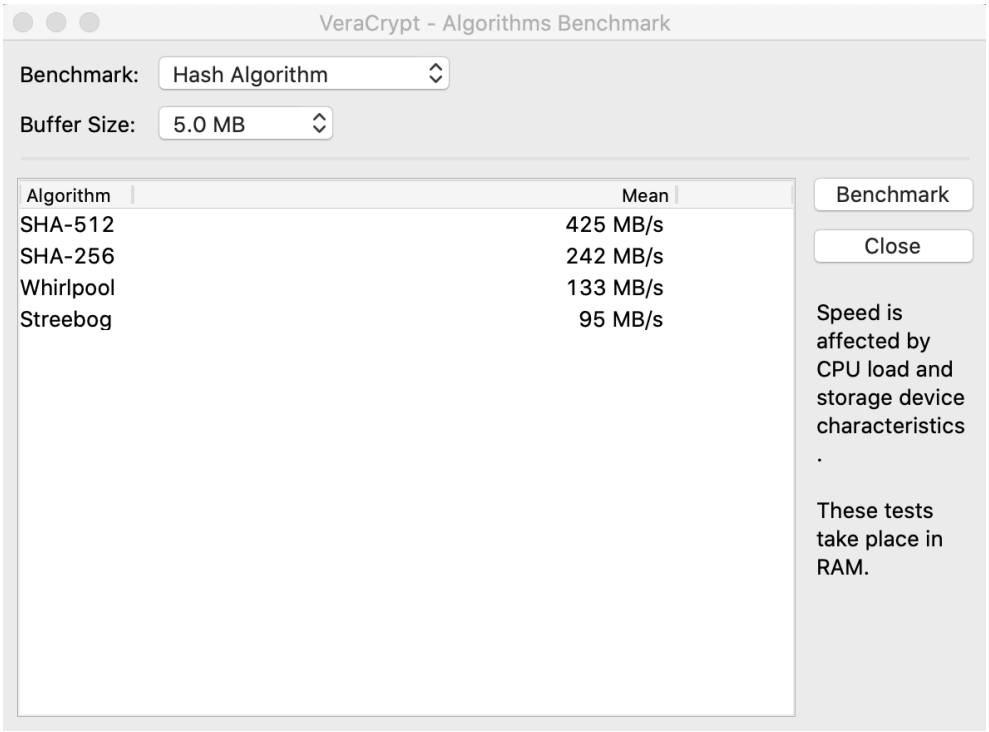


Fig. 31 Comparativa de algoritmos hash en VeraCrypt

Así, pues, a la hora de crear IDs de paquete y de talkgroup, se recibe como entrada el ID original y se devuelve como salida la cadena de 256 bits correspondiente a la entrada. Sin embargo, dos cadenas de este tamaño (ID de paquete e ID de talkgroup) harían relativamente larga la cabecera del paquete, por lo que se ha optado por recortar el hash. Es decir, del hash original solo se utilizará una parte. La longitud real de los IDs es, finalmente, de 32 bits (8 caracteres).

"hello world" string's SHA256 hash

ID bits	Discarded bits
B94D27B9	934D3E08A52E52D7DA7DABFAC484EFE37A5380EE9088F7ACE2EFCDE9

Fig. 32 Diagrama de obtención de ID a partir de un hash SHA256

Parecería de este modo que estamos rebajando la seguridad del hash considerablemente, pero de ninguna manera. El hash que se realiza es de 256 bits, por lo que sigue manteniendo el nivel de seguridad en caso de querer derivar el valor original a partir del hash. Lo único que se hace es obtener una parte de ese hash. Es cierto que de esta forma hay más probabilidades de que dos hashes coincidan para dos entradas diferentes, pero la probabilidad de que coincidan en el tiempo es remota.

8.3 Seguridad de la red WiFi

Para encriptar el tráfico de la red WiFi se ha optado por el sistema WPA2 (Wi-Fi Protected Access 2). La clave WPA2-Personal puede ser configurada por el usuario.

Esta encriptación provee protección ante sniffing de paquete, pero, en este caso, también sirve como medio de autenticación para poder usar el dispositivo. Como para poder usar el dispositivo es obligatorio conectarse a la zona WiFi que genera, se hace necesario, por consiguiente, proveer la clave WPA2 que la protege **[23]**.

9. Base de datos

La base de datos que se ha implementado se divide en dos partes. Por un lado, está la base de datos de usuario, almacenada en formato JSON y que es utilizada por el navegador para mostrar los datos del usuario en la interfaz gráfica. Por otro lado, está la base de datos de entorno, encargada de almacenar datos necesarios para el funcionamiento del ESP32 de forma rápida.

Toda esta información es almacenada en una base de datos SQLite 3, por medio de las funcionalidades que ofrece la librería ESP32 Arduino SQLite Lib (sqlite3.h) **[24]**. Es una librería que implementa las funciones principales de una base de datos SQLite 3, pero tiene muchas carencias que impiden el desarrollo de una base de datos optimizada. Por ejemplo, no se pueden crear relaciones ni claves principales o foráneas. En cualquier caso, para las necesidades del proyecto, ha resultado suficiente y ha mostrado buenos resultados en cuanto al tiempo de ejecución. Aquí el esquema de la base de datos:

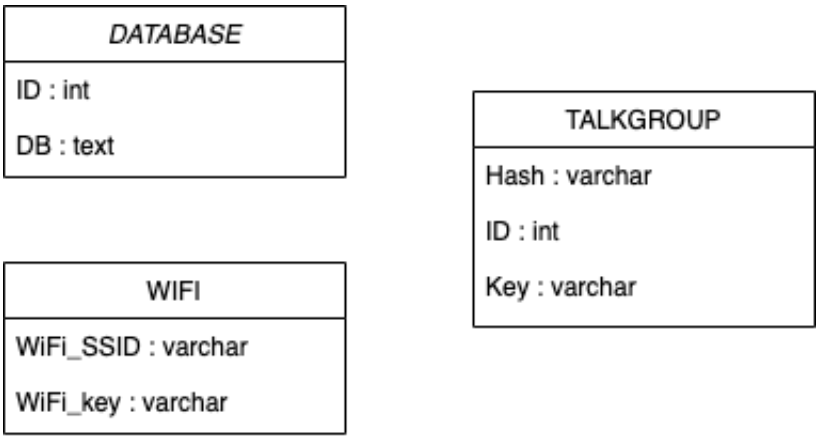


Fig. 33 Esquema de la base de datos

Como puede verse, no hay relaciones entre las tablas, porque la librería no lo permite. La información que se envía al navegador está almacenada en la tabla Database. En esta tabla únicamente se guardan un ID de usuario, que es espurio ya que no tiene ninguna función, pero que se ha implementado con vistas al futuro, donde una versión multiusuario del dispositivo pudiera desarrollarse, y también la base de datos en formato JSON. Esta base de datos se envía completa al navegador cuando el usuario se conecta. También es enviada completa cada vez que se hace un cambio de datos de importancia. Esos cambios son relativos a: SSID del WiFi, clave del WiFi, creado o borrado de un chat (talkgroup) y cambio de clave de un chat (talkgroup). En estos casos el envío al ESP32 para su guardado es necesario, porque afectan directamente al funcionamiento del dispositivo. Es decir, que cuando se realiza una de esas acciones enumeradas, los cambios se hacen primero en el resto de las tablas (por ejemplo, una actualización de clave de chat es directamente escrita en la tabla Talkgroup), y después se guarda la base de datos de usuario con los cambios realizado. De este modo, a falta

de relaciones, se consigue sincronizar todas las tablas en los puntos importantes. En el resto de los casos, la base de datos puede ser guardada manualmente a través de un botón en la interfaz gráfica y también es guardada automáticamente cada vez que se cierra la ventana del navegador. En todos estos casos la base de datos JSON es enviada al completo, lo que no es nada eficiente, pero es suficiente para un uso en el que la información de usuario no crece demasiado (un JSON con 30 chats y 5 mensajes en cada chat ronda los 60 kb).

La base de datos de usuario almacena la información de usuario (id, nombre), la lista de chats, los mensajes de cada chat, los contactos y los datos de wifi (SSID y clave). Esta base de datos tendría la siguiente estructura:

```
{
        "user_id": undefined,
        "username": undefined,
        "wifi_ssid": "LoRaChat",
        "wifi_key": "",
        "contacts": [],
        "chats": []
}
```

En las listas "contacts" y "chats" se guardan documentos JSON que representan cada entidad. Un contacto en formato JSON tendría esta forma:

```
{
        "id": userId,
        "name": name
}
```

Donde "id" es estrictamente el ID del contacto y "name" el nombre que el usuario elija para ese contacto. Un chat en formato JSON vendría descrito por:

```
{
        "id": chatId,
        "title": title,
        "key": key,
        "unread": 0,
        "id_counter": 0,
        "messages": []
}
```

Donde "id" es estrictamente el ID del talkgroup, "title" es el título que el usuario elija para el chat, "key" es la calve de cifrado/descifrado asociada al chat, "unread" es un contador de mensajes sin leer, "id_counter" es el contador de mensajes enviados por el usuario en ese chat (necesario para la creación de IDs de mensaje) y, finalmente, "messages" es una lista donde se almacenan los mensaje, que tendrían esta forma en JSON:

```
{
        "id": id,
        "mine": mine,
        "author": author,
        "text": text,
        "status": FLAG
}
```

Aquí, "id" es el id del mensaje, "mine" es un booleano que indica si el mensaje es del usuario local (true) o es de un contacto externo (false), "author" guarda el ID del autor del mensaje, "text" es el texto del mensaje y "status" es un entero que indica el estado del mensaje en base a estos tres valores, representados por variables globales:

MSG_PENDING = 0

Indica que el mensaje está pendiente de envío.

MSG_SENT = 1

Indica que el mensaje fue enviado correctamente por LoRa.

MSG_RECEIVED = 2

Indica que el mensaje fue recibido por alguien (ha llegado un ACK confirmándolo).

En cuanto al resto de tablas, la tabla de talkgroup almacena el ID hasheado del talkgroup, el ID original del talkgroup y la clave de cifrado del talkgroup.

La tabla WiFi almacena el SSID y la clave de la zona WiFi. Por defecto, el SSID es "LoRaChat" y la contraseña es vacía, lo que supone que la zona WiFi estará abierta al principio (para que el usuario se conecte y la proteja con una clave personal). También hay un campo llamado ID, cuya razón de existir es la misma que la que tenía en la tabla Database, es decir, por si en un futuro se habilita la funcionalidad multiusuario.

10. Pruebas

En este apartado únicamente se describirán las pruebas relacionadas con el funcionamiento del protocolo y de la red. Sobre las pruebas relacionadas con el funcionamiento del código y de GUI, limitarse a decir que han sido continuas en el tiempo, durante el desarrollo, y, una vez se consideró que existía una primera versión funcional del sistema, se hizo una auditoría completa dividida en los apartados ESP32 y GUI web. En el primer apartado, se comprobó que el funcionamiento no presentara errores y que, en caso de aparecer, fuese resiliente ante ellos. Se comprobó que el servidor web, el servidor de WebSockets, el módulo LoRa, la base de datos y el sistema de ficheros internos funcionaban según lo esperado. En el segundo apartado, relativo a la interfaz gráfica de usuario, se comprobó uno a uno que los casos de uso funcionasen correctamente, que los elementos gráficos no presentasen errores y que la interfaz actuase de forma correcta ante fallos de conexión.

En cuanto a las pruebas de red, se han querido testar varios factores. No obstante, ha habido muchos problemas a la hora de hacer pruebas, debido a la situación de excepcionalidad que se ha vivido estos últimos meses y la incapacidad de tener acceso al laboratorio de la Universidad. Solo se ha dispuesto de dos ESP32 propios para hacer todas las pruebas, lo que limita el alcance de éstas. Los factores que se han querido medir son: la distancia de funcionamiento efectivo, el correcto funcionamiento de la red mallada y el enrutamiento, y, por último, la carga que puede soportar la red y su relación con la pérdida de paquetes.

10.1 Distancia

Para medir la distancia, el procedimiento ha sido trivial. Se acordaron unos puntos del mapa de antemano y, una vez que la situación sanitaria permitió la salida al exterior, se ejecutaron unas breves medidas de distancia, progresivamente mayores.

Se hicieron pruebas en campo abierto a 200, 400, 600, 1000 y 1500 metros. La antena usada fue la que viene de fábrica con el ESP32. Sin embargo, al ser ésta genérica, no suele dar muy buenos resultados, por lo que se probó también el método tradicional de usar un simple cable como antena.

Los resultados a estas distancias fueron satisfactorios. En todos se conseguía recibir, aunque las pérdidas aumentaban con la distancia. A 1500 metros ya se empezaban a perder bastantes paquetes. Un factor importante fue el uso de CRC, pues de este modo se descartaban paquetes corruptos. Sin CRC llegaban más paquetes, pero algunos de ellos corruptos.

Finalmente, se realizó una prueba más ambiciosa, que abarcó los 5640 metros de distancia aproximadamente. Se consiguieron recibir paquetes a esa distancia, aunque con una probabilidad muy baja de éxito.

Fig. 34 Mapa de pruebas de distancia

También se ha llegado a probar en zona urbana, pero, como ya se estimaba de antes, los resultados con LoRa no son muy buenos cuando hay obstáculos de por medio. Se ha conseguido enviar a 150 metros entre casas, llegando rara vez los paquetes. De la misma forma, en área rural, abierta, a nada que existía un pequeño obstáculo como, por ejemplo, un árbol, la distancia de envío se reducía considerablemente.

Otro tipo de parámetros, como las interferencias y cuestiones más relacionadas con la radio, no se han podido probar por falta de instrumentos y material.

10.2 Funcionamiento de la malla

En el caso de las pruebas relativas al funcionamiento de la red y del enrutamiento, la limitación en cuanto a herramientas y la falta de acceso al laboratorio ha resultado en una dificultad para probar dichos parámetros. Al disponer únicamente de dos ESP32, se ha tenido que diseñar un escenario de forma rigurosa, así como documentar teóricamente los resultados que se esperaban de la práctica. Es decir, con una red "en malla" de dos nodos (A y B) y en base al protocolo LoRaChat, se ha descrito el intercambio de paquetes que debía acontecer ante el envío de un mensaje desde A hacia B, incluidos los paquetes ACK. El resultado esperado sería este:

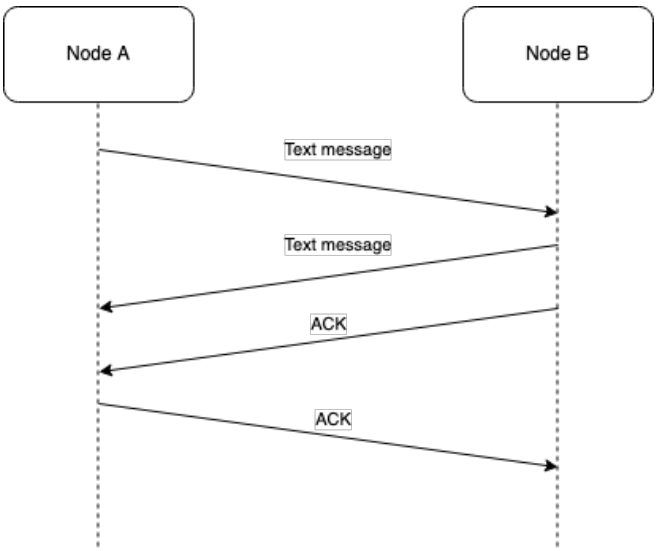


Fig. 35 Resultado esperado en pruebas de la malla LoRaChat con dos nodos

El nodo A envía un mensaje a el nodo B. Éste lo reenvía en tanto que cada nodo es también un rúter. Después, el nodo B también envía un mensaje ACK. Estos dos últimos mensajes llegarían al nodo A, que descartaría el primer mensaje por ser reciente (él mismo es su creador) y reenviaría el segundo mensaje, el ACK, por no ser reciente. Por último, el nodo B recibiría el ACK reenviado por el nodo A, descartándolo por ser reciente (él mismo es su creador).

Para comprobar el funcionamiento, se han imprimido por pantalla los paquetes recibidos.

Por parte del nodo A:

```
Sending TEXT: e1b7e475|0ffe1abd|o؟oFs؟q^؟؟؟؟؟G 3؟ ؟h؟- ;؟ ؟+؟
Received:     e1b7e475|0ffe1abd|o؟oFs؟q^؟؟؟؟؟G 3؟ ؟h؟- ;؟ ؟+؟
Received:     b8ad2bae|0ffe1abd| ;y؟ ؟ _J ؟# Q
Resending:    b8ad2bae|0ffe1abd| ;y؟ ؟ _J ؟# Q
```

Fig. 36 Output de las acciones en el nodo emisor en las pruebas de la malla LoRaChat con dos nodos

Por parte del nodo B:

```
Received:     e1b7e475|0ffe1abd|o؟oFs؟q^؟؟؟؟؟G 3؟ ؟h؟- ;؟ ؟+؟
Resending:    e1b7e475|0ffe1abd|o؟oFs؟q^؟؟؟؟؟G 3؟ ؟h؟- ;؟ ؟+؟
Sending ACK: b8ad2bae|0ffe1abd| ;y؟ ؟ _J ؟# Q
Received:     b8ad2bae|0ffe1abd| ;y؟ ؟ _J ؟# Q
```

Fig. 37 Output de las acciones en el nodo receptor en las pruebas de la malla LoRaChat con dos nodos

Como puede verse, el nodo A envía el paquete de mensaje de texto con ID "e1b7e475" que es recibido por B, reenviándolo al instante y, después, enviando un ACK con ID "b8ad2bae" para dar acuse de recibo. El nodo A volverá a recibir su propio mensaje reenviado por el nodo B (pero no lo reenviará por estar en la tabla de recientes) y el ACK que le confirma que su mensaje llegó al destino, el cual será reenviado por A y recibido por B.

En conclusión, se puede decir que la malla funciona perfectamente, si se dejan de lado las colisiones que, sin embargo, en la práctica siempre ocurren, como se verá en el capítulo 11.3.

10.3 Carga de la red

En cuanto a la medición de la carga que puede soportar la red y, una vez más, limitado el experimento por carencias técnicas, se decidió probar la cantidad de paquetes que se perdían según aumentaba la carga de la red.

Con el objetivo de probar esto, en un escenario de dos nodos (A y B), el nodo A enviaría mensajes al nodo B con una frecuencia progresivamente mayor, y en B se mediría el porcentaje de los paquetes recibidos. El nodo A se encargaría de enviar 10 paquetes de mensaje de texto, donde la longitud del mensaje y el retardo entre envíos irían cambiando y se irían combinando. Por tanto, se generaría un tráfico total de 40 paquetes en la red, entre mensajes de texto, ACKs y reenvíos.

Longitudes de mensajes: 30 y 300 caracteres.
Retardos entre envíos: 500, 400, 300, 200, 100, 60 y 40 ms.

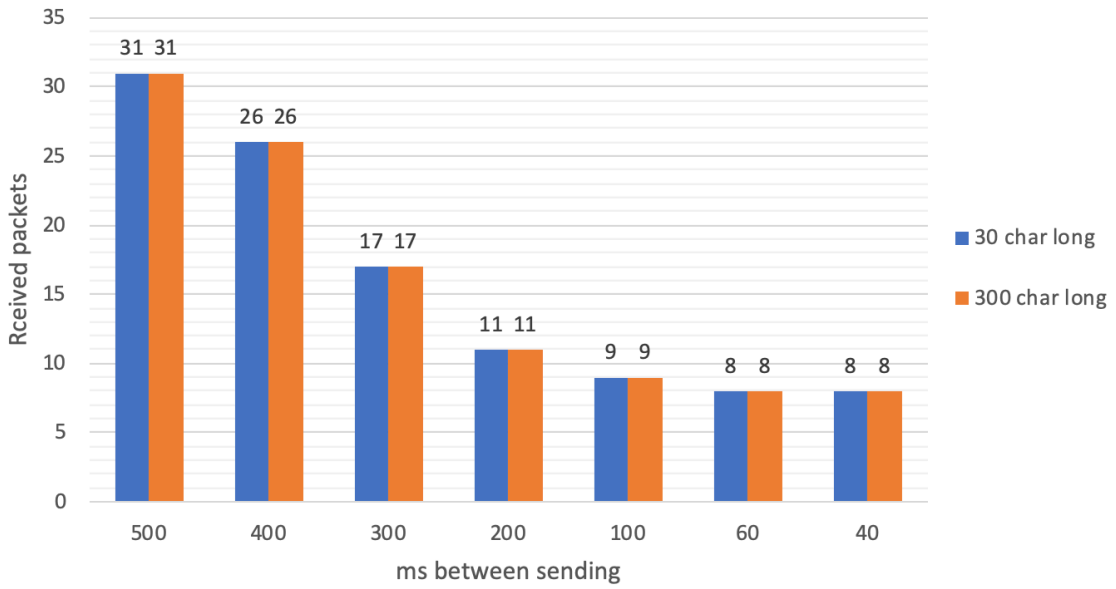


Fig. 38 Gráfica de rendimiento de la malla LoRaChat

Una de las primeras cosas que se ha notado durante los experimentos es que los resultados de éxito en la recepción de los paquetes no varían con diferentes tamaños de mensajes. Las pruebas se han hecho con más longitudes, pero al no variar, no se ha creído conveniente mostrar todos los datos, más allá de las que se han definido como las longitudes estándar de un mensaje corto (30 caracteres) y un mensaje mediano (300 caracteres).

Otra conclusión que se extrae del experimento es que la red no escala del todo bien cuando aumentan el tráfico, perdiéndose por el camino el 80% de los paquetes en los periodos entre envíos más cortos (60 y 40 ms). Esto supone también que la red no escalaría bien con el aumento de nodos, pues equivale a aumentar el tráfico cada vez que se envía un paquete. Por ejemplo, si en una red cinco nodos se alcanzan entre sí mutuamente, ante el envío de un mensaje todos los nodos tendrán que reenviar al menos una vez el mensaje y al menos una vez el ACK, por lo que cada nodo recibiría un total de diez paquetes.

10.4 Errores en la red

Al ver los resultados del experimento para medir la carga de red soportable, se ha realizado una hipótesis teórica para explicar la pérdida de paquetes, prescindiendo por completo de la medición y explicación de interferencias, la cual, como se ha dicho más arriba, no se ha podido medir por carencia de instrumentos.

Uno de los errores más comunes en todo tipo de red telemática es el causado por colisiones. Una colisión, grosso modo, ocurre cuando en un canal de información dos o más paquetes o datagramas colisionan entre sí, haciendo que toda o parte de la información se pierda. Existen múltiples métodos para controlar y solucionar estos errores, como Carrier Sense Multiple Access (CSMA). En el caso de LoRaChat, aún no se ha implementado un sistema de este tipo, por lo que la única forma de saber si un paquete ha llegado a su destino es mediante ACKs, los cuales, por cierto, también se pueden perder.

La hipótesis teórica para explicar la pérdida de paquetes por colisión, a continuación, se representa un escenario en el que hay 3 nodos: 1, 2 y 3. El nodo 1 alcanza a llegar al nodo 2, pero no al 3. El nodo 2, por su parte, alcanza al 1 y al 3. El nodo 3 alcanza únicamente al nodo 2. Los nodos 1 y 2 pertenecen al talkgroup para el que está dirigido el mensaje, mientras que el 3 no está suscrito a dicho talkgroup. Las acciones son resultado de un único envío por parte del nodo 1:

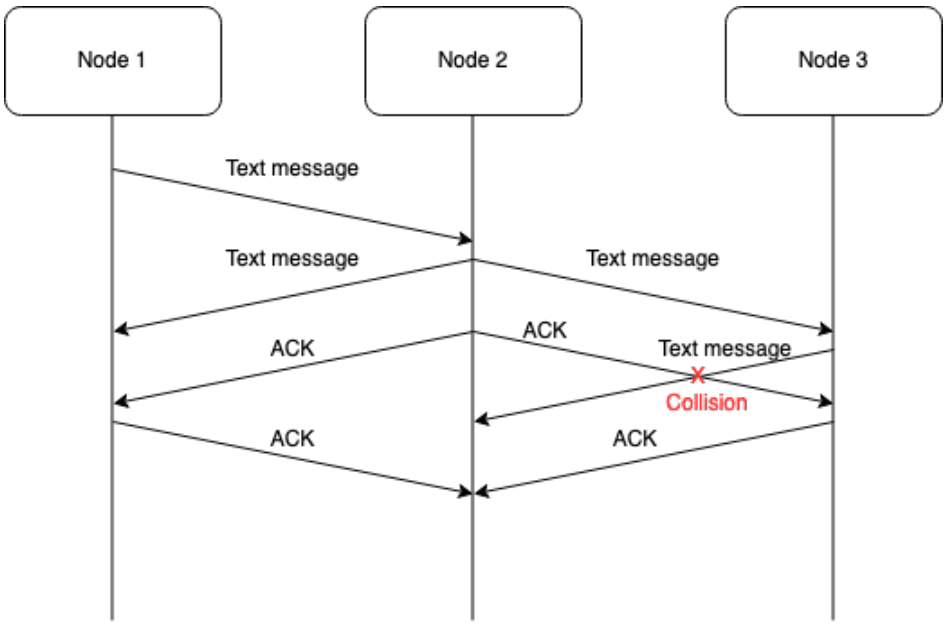


Fig. 39 Explicación hipotética para la pérdida de paquetes en la malla LoRaChat

Como puede verse, cada nodo se encarga de reenviar el mensaje de texto, además del ACK generado por el nodo 2. La colisión surge entre el nodo 2 y el nodo 3, al chocar el paquete ACK que envía el nodo 2 y el paquete de mensaje de texto reenviado por el nodo 3. En una malla de mayor envergadura, con más nodos y más tráfico, es de esperar que el número de estas colisiones aumente en la misma proporción.

Aunque esta es solo una hipótesis teórica, resulta factible teniendo en cuenta los tiempos en los que cada paquete se envía en teoría. No obstante, si esta coincidencia se diera siempre ocurrirían más colisiones de las que ocurren realmente. Además, cada paquete tarda en procesarse una cantidad diferente de tiempo, según otros factores como la ejecución que se esté realizando en el ESP32 o el tamaño del paquete.

11. Conclusión

Las conclusiones que se obtienen después del desarrollo del trabajo son que, por un lado, se ha conseguido aplicar de forma efectiva una solución a la problemática que se pretendía solucionar y, por otro lado, la solución podría mejorarse bastante en cuanto al funcionamiento de la red.

En lo que se refiere al dispositivo como tal, se ha logrado crear en una única pieza de hardware un sistema de usuario universal. Es decir, se ha conseguido un sistema portátil capaz de ser usado con cualquier dispositivo con una pantalla, la capacidad de conectarse a una red WiFi y que pueda ejecutar un navegador web. Así como otro tipo de soluciones, ésta que se presenta en el trabajo realizado ofrece a los usuarios independencia respecto de otras aplicaciones o programas. También se ha conseguido desarrollar una interfaz de usuario cómoda y bastante parecida a interfaces de servicios de mensajería conocidos. La seguridad ha sido un factor muy tenido en cuenta, implementándose encriptación simétrica para que, todos aquellos participantes de un talkgroup puedan hablar de forma segura.

Se ha desarrollado un protocolo de red con topología de malla desde cero, con la mira puesta en el fácil y rápido despliegue de los nodos, y con seguridad y encriptación añadida por defecto a nivel de protocolo. En el protocolo LoRaChat, cada usuario se hace partícipe de la extensión de la red y la dependencia de terceros, como servidores o rúters centrales, se ha eliminado completamente. Las tareas relativas al protocolo son, con toda seguridad, las que más esfuerzo han requerido, por el hecho de tener que diseñar el protocolo sin apenas apoyarse en información, ya que la gran mayoría, si no todos los protocolos de enrutamiento en malla utilizan direcciones y, por tanto, tienen la capacidad de crear enlaces y de calcular rutas con eficacia.

Sin embargo, el proyecto ha presentado problemas en la práctica, en cuanto al rendimiento de la red se refiere. Teniendo en cuenta que el protocolo ha sido diseñado para funcionar sin direcciones de nodo, no se ha podido implementar un algoritmo de cálculo de rutas óptimas. Tampoco se han podido establecer enlaces lógicos entre los nodos, así como tampoco se han podido direccionar los mensajes hacia un nodo concreto. El modo de operación es un continuo broadcast en el que todos reciben y reenvían los mensajes y solo procesan aquellos que pertenecen a un talkgroup al que el nodo está suscrito. De esta manera, la red se satura de manera tal vez excesiva. Tampoco se han implementado canales (LoRa lo permite), ya que, al no existir enlaces, no parecen ser de mucha utilidad. No obstante, del mismo modo que existen suscripciones a un conjunto de talkgroups determinado, también podría haberse implementado un sistema mediante el cual el usuario sería capaz de elegir el canal en el que quiere operar **[25]**. Otra solución habría sido implementar algún sistema de prevención de colisiones.

En conclusión, si bien no se ha conseguido crear un servicio de mensajería con un buen funcionamiento en una red amplia, sí que se cree haber logrado implementar un sistema de comunicación por radio totalmente funcional, seguro y con una aceptable comodidad de despliegue y usabilidad de cara al usuario en situaciones donde el flujo de

información se ve cortado por algún acontecimiento anormal. Incluso fuera de este tipo de situaciones, su uso también puede ser de utilidad.

12. Líneas futuras

La tarea primordial para el futuro de este proyecto es hacer más pruebas de la red y los dispositivos. Se trataría de ir comprobando hasta qué punto es eficiente la red cuando se aumenta el número de nodos y el tráfico. Además, habría que realizar más pruebas de distancia con diferentes configuraciones de antenas. En resumen, probar el sistema de forma exhaustiva para mejorar la calidad de servicio.

Otras ideas de futuro para el proyecto pueden encontrarse en la siguiente lista:

- Optimizar el código para que el procesamiento de los paquetes se desarrolle con mayor rapidez, en aras de mejorar el rendimiento de la red.

- Permitir al usuario elegir el canal en el que operar.

- Mejorar el sistema de base de datos para optimizar la rapidez de ejecución de consultas y transmisión de información.

- Añadir la característica multiusuario, para que el dispositivo pueda ser usado por varios usuarios a la vez.

- Realizar una auditoría de seguridad.

- Probar configuraciones de potencia y antenas para alcanzar mayores distancias.

- Reducir el consumo de energía del dispositivo. Utilizar funciones como *deep sleep*.

- Crear un sistema para facilitar el acceso a la interfaz gráfica mediante el navegador. Esto se logró programando un servidor DNS y un portal captivo. Sin embargo, no llego a convencer porque en algunos dispositivos los portales captivos no producen una buena experiencia de usuario.

- Diseñar una carcasa 3D para envolver el ESP32 y una batería de alimentación. Esta idea estaba pensada para realizarla en el trabajo, pero la crisis sanitaria ha impedido el acceso a una impresora 3D. En cualquier caso, el ESP32 permite conectarle una batería y recargarla a través del conector micro USB de la placa. De este modo, se tendría un dispositivo compacto y con la capacidad de funcionar sin estar enchufado a una fuente de alimentación estática, además de poder recargar la batería de forma fácil con cualquier cargador micro USB.

13. Difusión

El proyecto ha sido publicado en un repositorio de GitHub, donde se ha explicado sobre qué trata LoRaChat y se ha intentado explicar de la forma más sencilla posible cómo instalar las librerías y el entorno para cargar el código en el ESP32. Desde el inicio también se ha mantenido el enfoque en la facilidad de uso y en la comprensión del código para que cualquiera pueda sugerir mejoras o realizar sus implementaciones propias. Finalmente, el código ha sido distribuido bajo licencia GNU General Public License versión 3.

Enlace al repositorio: https://github.com/commodore1917/lorachat

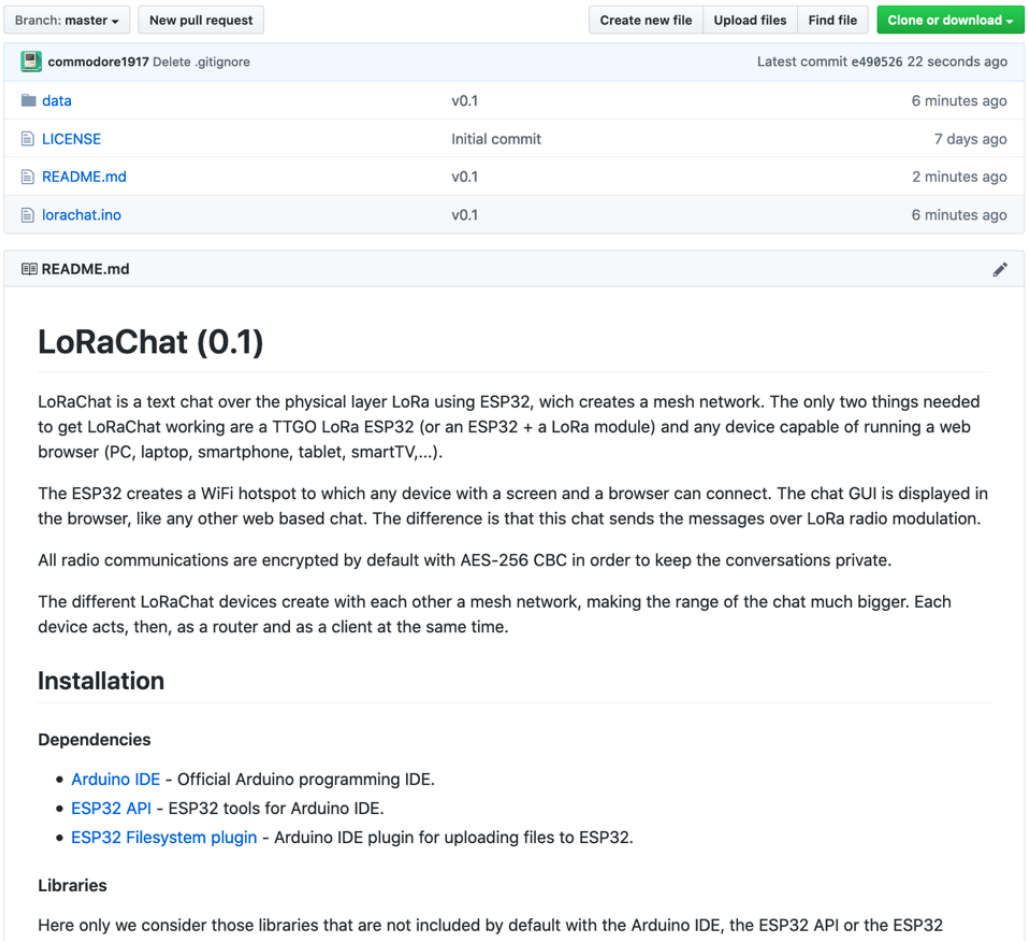


Fig. 40 Vista principal del repositorio de GitHub de LoRaChat

14. Referencia de imágenes

15. Bibliografía y referencias

[1] http://www.servalproject.org/

[2] https://www.bbc.com/news/technology-27225869

[3] https://briarproject.org/

[4] https://code.briarproject.org/briar/briar-spec/blob/master/protocols/BTP.md

[5] https://www.hackster.io/scottpowell69/lora-mesh-chat-5267d9

[6] https://github.com/spleenware/ripple

[7] https://www.hackster.io/akarsh98/lora-messenger-for-two-devices-for-distances-up-to-8km-f2dc66

[8] https://github.com/Damianuscz/Lora-Chat-Device

[9] https://xmpp.org

[10] https://www.mickmake.com/post/sms-lora-long-distance-sms-without-4g-project/

[11] https://alfaiot.com/blog/ultimas-noticias-2/post/que-es-lora-2

[12] https://en.wikipedia.org/wiki/Chirp_spread_spectrum

[13] https://lora-alliance.org/about-lorawan

[14] https://lora-alliance.org/sites/default/files/2018-04/what-is-lorawan.pdf

[15] https://hackernoon.com/9-things-you-need-to-know-about-mesh-networks-f61a77e5751a

[16] https://www.taitradioacademy.com/topic/what-is-dmr-1/

[17] https://downloads.open-mesh.org/batman/papers/batman-adv_network_coding.pdf

[18] https://mentor.ieee.org/802.11/public/06/11-06-1778-01-000s-hwmp-specification.doc

[19] https://www.semtech.com/products/wireless-rf/lora-transceivers/sx1276

[20] https://pdfs.semanticscholar.org/5711/556af6e5edcc4432dc7a48fc5a3688bc3612.pdf

[21] https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

[22] https://cheapsslsecurity.com/blog/what-is-sha2-and-what-are-sha-2-ssl-certificates/

[23] https://www.howtogeek.com/167783/htg-explains-the-difference-between-wep-wpa-and-wpa2-wireless-encryption-and-why-it-matters/

[24] https://github.com/siara-cc/esp32_arduino_sqlite3_lib

[25] https://www.rfwireless-world.com/Tutorials/LoRa-channels-list.html

16. Código fuente

En este apartado no se incluye el código CSS, por ser escrito en su gran mayoría por w3schools.com y al que se han añadido unas pocas reglas de estilo propias. En cualquier caso, el código completo puede encontrarse en el repositorio de GitHub de LoRaChat.

16.1 Código Arduino

```
/*********************************************************
 *
 * LoRaChat.
 * Chat over LoRa.
 * 2020.
 *
 *********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>
#include <WiFi.h>
#include <WebSocketsServer.h>
#include <ESPAsyncWebServer.h>
#include <Wire.h>
#include <SPI.h>
#include <LoRa.h>
#include <FS.h>
#include <SPIFFS.h>
#include <mbedtls/md.h>
#include <hwcrypto/aes.h>

#define FORMAT_SPIFFS_IF_FAILED true

#define SS   18
#define RST  14
#define DIO0 26
#define FREQ 868E6

#define LORA_PACKET_ID_SIZE 8
#define LORA_TG_ID_SIZE 8

#define ROUTING 1
#define ROUTING_TABLE_SIZE 4

#define RECEIVED_BUFFER_SIZE 5

/*** LORACHAT CODES ***/
#define LORA_MSG 0
#define LORA_ACK 1
/**********************/

/*** WEBSOCKET CODES ***/
#define WS_ERROR -1
#define WS_SEND_MSG 0
#define WS_RECV_MSG 1
#define WS_RECV_ACK 2
#define WS_MSG_SENT 3
```

```
#define WS_DB_REQ 4
#define WS_DB_SEND 5
#define WS_DB_RECV 6
#define WS_ADD_CHAT 7
#define WS_DEL_CHAT 8
#define WS_SET_CHAT_KEY 9
#define WS_SET_WIFI_SSID 10
#define WS_SET_WIFI_KEY 11
#define WS_BUFFERED_MSG 12
/**********************/


typedef struct RoutingTable{
     String table[ROUTING_TABLE_SIZE];
     int last = 0;
}RoutingTable;

typedef struct ReceivedBuffer{
     String buf[RECEIVED_BUFFER_SIZE];
     int n = 0;
}ReceivedBuffer;

RoutingTable routingTable; // Storage for recently received packet's
hashes
ReceivedBuffer messageBuffer; // Buffer for storing received messages
when the client is not connected

WebSocketsServer webSocket = WebSocketsServer(81); // Servidor
websocket: 192.168.4.1:81
AsyncWebServer server(80); // Servidor web: 192.168.4.1:80

sqlite3 *db; // Database
String db_response = ""; // Database query response

bool clientConnected = false;

void setup() {

     Serial.begin(9600);

     if(!SPIFFS.begin(FORMAT_SPIFFS_IF_FAILED))
Serial.println("[ERROR] SPIFFS Mount Failed");
     else Serial.println("[OK] SPIFFS active");

     sqlite3_initialize(); // Initialize SQLite engine
     //db_removeDatabase("lorachat.db");
     if(db_open("/spiffs/lorachat.db", &db)) Serial.println("[ERROR]
Database init failed");
     else Serial.println("[OK] Database active");
     if(!db_check(db)) db_create(db); // If the database does not
exist, creates it

     initWifiAP();
     Serial.println("[OK] WiFi AP active");

     initWebSocket();
     Serial.println("[OK] WebSocket server active");

     initWebServer();
     Serial.println("[OK] Web server active");
```

```
      initLoRa();
      Serial.println("[OK] LoRa active");

}

void loop() {

      onReceive(LoRa.parsePacket()); // Recibir paquete de LoRa

      webSocket.loop(); // Gestionar clientes del web socket

}



/*********************************************************/
/*                          WIFI                         */
/*********************************************************/

/**
 * WiFi AP setup and initialization.
 */
void initWifiAP() {
      /**
    * Set up an access point
    * @param ssid               Pointer to the SSID (max 63 char).
    * @param passphrase         (for WPA2 min 8 char, for open use NULL)
    * @param channel            WiFi channel number, 1 - 13.
    * @param ssid_hidden        Network cloaking (0 = broadcast SSID, 1
= hide SSID)
    * @param max_connection     Max simultaneous connected clients, 1 -
4.
   */
      String ssid_aux = db_getWifiSSID(db);
      char ssid_char[ssid_aux.length()+1];
      ssid_aux.toCharArray(ssid_char, ssid_aux.length()+1);
      const char* ssid = (const char*)ssid_char;
      String pass_aux = db_getWifiKey(db);
      char pass_char[pass_aux.length()+1];
      pass_aux.toCharArray(pass_char, pass_aux.length()+1);
      const char* pass = (const char*)pass_char;
      WiFi.softAP(ssid, pass, 6, 0, 1);
}



/*********************************************************/
/*                          Web Server                   */
/*********************************************************/

/**
 * Configures and intializes the web server.
 */
void initWebServer() {
      server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
        request->send(SPIFFS, "/index.html", "text/html");
      });
```

```
      server.on("/lorachat.css", HTTP_GET, [](AsyncWebServerRequest
*request){
        request->send(SPIFFS, "/lorachat.css", "text/css");
      });

      server.on("/lorachat.js", HTTP_GET, [](AsyncWebServerRequest
*request){
        request->send(SPIFFS, "/lorachat.js", "text/javascript");
      });

      server.begin(); // Begin web server
}




/*************************************************************/
/*                          WebSocket                        */
/*************************************************************/

/**
 * Handles with messages received over WebSocket.
 */
void webSocketMessageHandler(const char * msg) {

      const char * delim = "|";

      int type = atoi(strtok((char*)msg, delim));

      switch(type) {

        case WS_SEND_MSG: // Received LoRa message to send
        {
          int chatId = atoi(strtok(NULL, delim));
          int msgId = atoi(strtok(NULL, delim));
          int author = atoi(strtok(NULL, delim));
          char* messageText = strtok(NULL, delim);
          if(sendLoRaTextMessage(chatId, author, msgId, messageText))
            webSocketMsgSent(chatId, msgId, author); // Only if the
packet was send.
        } break;

        case WS_DB_REQ: // Database requested
        {
          webSocketSendDatabase(db_getUserDB(db)); // send db
        } break;

        case WS_DB_SEND: // Database received
        {
          char* userDB = strtok(NULL, delim);
          db_setUserDB(db, userDB); // save db
          webSocketDatabaseReceived();
        } break;

        case WS_ADD_CHAT: // Add chat received
        {
          int chatId = atoi(strtok(NULL, delim));
          char* chatKey = strtok(NULL, delim);
          db_insertTG(db, chatId, chatKey);
```

```
        } break;

        case WS_DEL_CHAT: // Delete chat received
        {
          int chatId = atoi(strtok(NULL, delim));
          db_deleteTG(db, chatId);
        } break;

        case WS_SET_CHAT_KEY: // Set chat key received
        {
          int chatId = atoi(strtok(NULL, delim));
          char* chatKey = strtok(NULL, delim);
          db_setTGKey(db, chatId, chatKey);
        } break;

        case WS_SET_WIFI_SSID: // Set wifi SSID received
        {
          char* ssid = strtok(NULL, delim);
          db_setWifiSSID(db, ssid);
        } break;

        case WS_SET_WIFI_KEY: // Set wifi key received
        {
          char* key = strtok(NULL, delim);
          db_setWifiKey(db, key);
        } break;

        case WS_BUFFERED_MSG: // Send buffered messages
        {
          if(bufferedMessages()) {
            webSocketSendBufferedMessages();
            freeMessageBuffer();
          }
        } break;

        default: break;
      }

}

/**
 * Generates a text message json to send it over websocket.
 */
String webSocketGenerateTextMessageJson(int talkgroupId, int
messageId, int author, String messageText) {
      String json = "";
      json += "{\"type\":" + String(WS_RECV_MSG);
      json += ",\"chatId\":" + String(talkgroupId);
      json += ",\"msgId\":" + String(messageId);
      json += ",\"author\":" + String(author);
      json += ",\"text\":\"" + String(messageText);
      json += "\"}";
      return json;
}

/**
 * Sends a ws message from a LoRa received text message.
 */
void webSocketSendTextMessage(int talkgroupId, int messageId, int
author, String messageText) {
```

```
	String json = webSocketGenerateTextMessageJson(talkgroupId,
messageId, author, messageText);
	webSocket.broadcastTXT(json);
}

/**
 * Sends a ws message from a LoRa received ACK.
 */
String webSocketGenerateACKJson(int talkgroupId, int messageId, int
author) {
	String json = "";
	json += "{\"type\":" + String(WS_RECV_ACK);
	json += ",\"chatId\":" + String(talkgroupId);
	json += ",\"msgId\":" + String(messageId);
	json += ",\"author\":" + String(author);
	json += "}";
	return json;
}

/**
 * Sends a ws message from a LoRa received ACK.
 */
void webSocketSendACK(int talkgroupId, int messageId, int author) {
	String json = webSocketGenerateACKJson(talkgroupId, messageId,
author);
	webSocket.broadcastTXT(json);
}

/**
 * Confirms that a LoRa message has been sent.
 */
void webSocketMsgSent(int talkgroupId, int messageId, int author) {
	String json = "";
	json += "{\"type\":" + String(WS_MSG_SENT);
	json += ",\"chatId\":" + String(talkgroupId);
	json += ",\"msgId\":" + String(messageId);
	json += ",\"author\":" + String(author);
	json += "}";
	webSocket.broadcastTXT(json);
}

/**
 * Confirms that the database has been received.
 */
void webSocketDatabaseReceived() {
	String json = "";
	json += "{\"type\":" + String(WS_DB_RECV);
	json += "}";
	webSocket.broadcastTXT(json);
}

/**
 * Sends user's database over the web socket.
 */
void webSocketSendDatabase(String database) {
	String json = "";
	json += "{\"type\":" + String(WS_DB_SEND);
	json += ",\"db\":" + database;
	json += "}";
	webSocket.broadcastTXT(json);
}
```

```
/**
 * Sends buffered messages to client.
 */
void webSocketSendBufferedMessages() {
     for(int i=0; i<messageBuffer.n; i++) {
       webSocket.broadcastTXT(messageBuffer.buf[i]);
     }
}

/**
 * WebSocket event handler.
 */
void onWebSocketEvent(uint8_t num, WStype_t type, uint8_t * payload,
size_t length) {
     switch(type) {

       case WStype_DISCONNECTED: // When client disconnects
         clientConnected = false;
         Serial.printf("[%u] Client disconnected!\n", num);
         break;

       case WStype_CONNECTED: // When client connects
         {
           clientConnected = true;
           IPAddress ip = webSocket.remoteIP(num);
           Serial.printf("[%u] Connection from ", num);
           Serial.println(ip.toString());
         }
         break;

       case WStype_TEXT: // Text
         {
           //Serial.printf("[%u] get Text: %s\n", num, payload);
           webSocketMessageHandler((const char *)payload);
         }
         break;

       default: break;
     }
}

/**
 * Initializes WebSocket server.
 */
void initWebSocket() {
     webSocket.begin();
     webSocket.onEvent(onWebSocketEvent);  // Assign handler to event
listener
}




/*******************************************************/
/*                        LoRa                         */
/*******************************************************/

/**
 * Initializes LoRa module.
```

```
 */
void initLoRa(){
     SPI.begin(SCK, MISO, MOSI, SS);
     LoRa.setPins(SS, RST, DIO0);
     LoRa.setTxPower(20); // Max is 20
     LoRa.enableCrc();
     if (!LoRa.begin(FREQ)) while (1); // 433 Mhz
}

/**
 * Sends a message over LoRa.
 */
int sendLoRaTextMessage(int talkgroupId, int author, int messageId,
String messageText) {

     String barePacket = String(talkgroupId) + "|";
     barePacket += String(LORA_MSG) + "|";
     barePacket += String(author) + "|";
     barePacket += String(messageId) + "|";
     barePacket += messageText;
     String packetHash = packetHashedId(barePacket);
     String tgHash = talkgroupHashedId(String(talkgroupId));
     String encryptedPacket = str_AES256_enc(db_getTGKeyById(db,
talkgroupId), tgHash, barePacket); // Get key and encrypt

     int b = LoRa.beginPacket();
     LoRa.print(packetHash);
     LoRa.print("|");
     LoRa.print(tgHash);
     LoRa.print("|");
     LoRa.print(encryptedPacket);
     int e = LoRa.endPacket();

     if(ROUTING) insertPacketIntoRecents(packetHash);

     return b && e;
}

/**
 * Sends an ACK of a received message over LoRa.
 */
int sendLoRaACK(int talkgroupId, int author, int messageId) {

     String barePacket = String(talkgroupId) + "|";
     barePacket += String(LORA_ACK) + "|";
     barePacket += String(author) + "|";
     barePacket += String(messageId);
     String packetHash = packetHashedId(barePacket);
     String tgHash = talkgroupHashedId(String(talkgroupId));
     String encryptedPacket = str_AES256_enc(db_getTGKeyById(db,
talkgroupId), tgHash, barePacket); // Get key and encrypt

     int b = LoRa.beginPacket();
     LoRa.print(packetHash);
     LoRa.print("|");
     LoRa.print(tgHash);
     LoRa.print("|");
     LoRa.print(encryptedPacket);
     int e = LoRa.endPacket();

     if(ROUTING) insertPacketIntoRecents(packetHash);
```

```
	return b && e;
}

/***
 * Resends a packet over LoRa.
 */
int resendLoRaPacket(String packet) {

	int b = LoRa.beginPacket();
	LoRa.print(packet);
	int e = LoRa.endPacket();

	return b && e;
}



/**
 * Performed when a LoRa packet is received.
 */
void onReceive(int packetSize) {

	if(!packetSize) return;

	// Get bare packet
	char packet[packetSize];
	char packet_copy[packetSize];
	int i = 0;
	while (LoRa.available()) { packet[i++] = (char)LoRa.read();}

	strcpy(packet_copy, packet);

	const char * delim = "|";

	// Extract packet id
	char* packetHashId = strtok(packet, delim);

	// Checks size of the packet id
	if(strlen(packetHashId) != LORA_PACKET_ID_SIZE) return;

	// Check if the packet is recent
	if(isPacketInRecents(String(packetHashId))) return; // If it is,
ignore it

	// Resend
	if(ROUTING) resendLoRaPacket(packet_copy);

	// Insert packet into recents
	if(ROUTING) insertPacketIntoRecents(String(packetHashId));

	// Extract talkgroup id
	char* receivedTgHash = strtok(NULL, delim);

	// Checks size of the talkgroup id
	if(strlen(receivedTgHash) != LORA_TG_ID_SIZE) return;

	// Check if the tg is in my database
	if(!db_existsTGHash(db, receivedTgHash)) return; // If not, it is
not for me
```

```
      char* encryptedPacket = strtok(NULL, delim);

      // Decrypt packet
      String decryptedPacket = str_AES256_dec(db_getTGKeyByHash(db,
receivedTgHash), receivedTgHash, encryptedPacket); // Get key and
encrypt

      // Check if decrypted packet hash id and received packet hash id
are the same
      if(packetHashedId(decryptedPacket) != String(packetHashId))
return; // If not the same, break

      // Extract data from decrypted packet
      char dec[decryptedPacket.length()+1];
      decryptedPacket.toCharArray(dec, decryptedPacket.length()+1);
      int tgId = atoi(strtok(dec, delim));

      // Check if real tgHash and packet's tgHash are the same
      if(talkgroupHashedId(String(tgId)) != receivedTgHash) return; //
If not the same, break

      int msgType = atoi(strtok(NULL, delim));
      int author = atoi(strtok(NULL, delim));
      int msgId = atoi(strtok(NULL, delim));

      switch(msgType) { // Message type actions

        case LORA_MSG: // Text message
        {
          sendLoRaACK(tgId, author, msgId); // Send ACK
          String text = String(strtok(NULL, delim));
          if(clientConnected) // if the client is connected
            webSocketSendTextMessage(tgId, msgId, author, text); //
send message over ws
          else { // if the client is not connected
            String json = webSocketGenerateTextMessageJson(tgId, msgId,
author, text);
            insertMessageInBuffer(json); // insert message into the
buffer
          }
        } break;

        case LORA_ACK: { // Message ACK
          if(clientConnected) // if the client is connected
            webSocketSendACK(tgId, msgId, author); // send ack over ws
          else { // if the client is not connected
            String json = webSocketGenerateACKJson(tgId, msgId,
author);
            insertMessageInBuffer(json); // insert message into the
buffer
          }
        } break;

        default: break;
      }

}

/**
 * Returns the ID of a LoRa packet.
 */
```

```
String packetHashedId(String barePacket) {
     return strToSHA256(barePacket).substring(0, LORA_PACKET_ID_SIZE);
}

/**
 * Returns the hashed ID of a talkgroup.
 */
String talkgroupHashedId(String tgId) {
     return strToSHA256(tgId).substring(0, LORA_TG_ID_SIZE);
}




/*****************************************************/
/*                        SQLite                     */
/*****************************************************/

const char* data = "Callback function called";

/**
 * SQLite callback function.
 */
static int callback(void *data, int argc, char **argv, char
**azColName) {
      int i;
      for (i = 0; i<argc-1; i++) {
         db_response += argv[i] ? argv[i] : "null";
         db_response += "/";
      }
      db_response += argv[argc-1] ? argv[argc-1] : "null";
      db_response += "|";
      //Serial.println(db_response);
      return 0;
}

/**
 * Opens SQLite database.
 * Returns: 1 = error; 0 = ok;
 */
int db_open(const char *filename, sqlite3 **db) {
      int rc = sqlite3_open(filename, db);
      return rc;
}

/**
 * Checks if the database is created -> 0 = no, >0 = yes.
 */
int db_check(sqlite3 *db) {
     db_response = "";
     db_exec(db, "SELECT count(*) FROM sqlite_master WHERE type =
'table';");
     db_response.remove(db_response.length()-1);
     return db_response.toInt();
}

/**
 * Creates and intializes tables on the database.
 */
int db_create(sqlite3 *db) {
```

```
      db_exec(db, "CREATE TABLE TALKGROUP(HASH VARCHAR(8), ID INT, KEY
VARCHAR(64));");
      db_exec(db, "CREATE TABLE WIFI(ID INT, WIFI_SSID VARCHAR(32)
DEFAULT 'LoraChat', WIFI_KEY VARCHAR(32) DEFAULT '');");
      db_exec(db, "INSERT INTO WIFI (ID) VALUES (1);");
      db_exec(db, "CREATE TABLE DATABASE(ID INT, DB TEXT);");
      db_exec(db, "INSERT INTO DATABASE (ID, DB) VALUES (1,
'{\"user_id\": \"\",\"username\": \"LoRaChatter\",\"wifi_ssid\":
\"LoRaChat\",\"wifi_key\": \"\",\"contacts\": [],\"chats\": []}');");
}

char *zErrMsg = 0;

/**
 * SQLite querys executor.
 */
int db_exec(sqlite3 *db, const char *sql) {
      //Serial.println(sql);
      long start = micros();
      int rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
      if (rc != SQLITE_OK) {
          Serial.printf("SQL error: %s\n", zErrMsg);
          sqlite3_free(zErrMsg);
      }
      //Serial.print(F("Time taken:"));
      //Serial.println(micros()-start);
      return rc;
}

/**
 * Removes the database from the storage.
 */
void db_removeDatabase(String filename) {
      SPIFFS.remove("/" + filename);
      SPIFFS.remove("/" + filename + "-journal");
}

/**
 * Sets the user's database (JSON) on SQLite database.
 */
int db_setUserDB(sqlite3 *db, String userDB) {
      String sql = "UPDATE DATABASE SET DB = '" + userDB + "' WHERE ID
= 1;";
      return db_exec(db, sql.c_str());
}

/**
 * Gets the user's database (JSON) from SQLite database.
 */
String db_getUserDB(sqlite3 *db) {
      db_response = "";
      db_exec(db, "SELECT DB FROM DATABASE WHERE ID = 1;");
      db_response.remove(db_response.length()-1);
      return db_response;
}

/**
 * Inserts a new TG into the SQLite database.
 */
int db_insertTG(sqlite3 *db, int tgId, String tgKey) {
```

```
	String sql = "INSERT INTO TALKGROUP (HASH, ID, KEY) VALUES ('" +
talkgroupHashedId(String(tgId))  + "', " + String(tgId) + ", '" +
tgKey + "');";
	return db_exec(db, sql.c_str());
}

/**
 * Deletes a TG from the SQLite database.
 */
int db_deleteTG(sqlite3 *db, int tgId) {
	String sql = "DELETE FROM TALKGROUP WHERE ID = " + String(tgId) +
";";
	return db_exec(db, sql.c_str());
}

/**
 * Gets the hash id of a talkgroup from the SQLite database.
 */
String db_getTGHash(sqlite3 *db, int tgId) {
	db_response = "";
	String sql = "SELECT HASH FROM TALKGROUP WHERE ID = " +
String(tgId) + ";";
	db_exec(db, sql.c_str());
	db_response.remove(db_response.length()-1);
	return db_response;
}

/**
 * Gets all TGs info from the SQLite database.
 */
String db_getTGs(sqlite3 *db) {
	db_response = "";
	String sql = "SELECT * FROM TALKGROUP;";
	db_exec(db, sql.c_str());
	db_response.remove(db_response.length()-1);
	return db_response;
}

/**
 * Checks if a TG hash exists in the SQLite database.
 * Returns: if exists = 1, else = 0.
 */
int db_existsTGHash(sqlite3 *db, String hash) {
	db_response = "";
	String sql = "SELECT COUNT(*) FROM TALKGROUP WHERE HASH = '" +
String(hash) + "';";
	db_exec(db, sql.c_str());
	db_response.remove(db_response.length()-1);
	return db_response.toInt();
}

/**
 * Sets the key of a talkgroup into the SQLite database.
 */
int db_setTGKey(sqlite3 *db, int tgId, String tgKey) {
	String sql = "UPDATE TALKGROUP SET KEY = '" + String(tgKey) + "'
WHERE ID = " + String(tgId) + ";";
	return db_exec(db, sql.c_str());
}

/**
```

```
 * Gets the key of a talkgroup by ID from the SQLite database.
 */
String db_getTGKeyById(sqlite3 *db, int tgId) {
     db_response = "";
     String sql = "SELECT KEY FROM TALKGROUP WHERE ID = " +
String(tgId) + ";";
     db_exec(db, sql.c_str());
     db_response.remove(db_response.length()-1);
     return db_response;
}

/**
 * Gets the key of a talkgroup by hsh from the SQLite database.
 */
String db_getTGKeyByHash(sqlite3 *db, String hash) {
     db_response = "";
     String sql = "SELECT KEY FROM TALKGROUP WHERE HASH = '" +
String(hash) + "';";
     db_exec(db, sql.c_str());
     db_response.remove(db_response.length()-1);
     return db_response;
}

/**
 * Sets the wifi key into the SQLite database.
 */
int db_setWifiKey(sqlite3 *db, String wifiKey) {
     String sql = "UPDATE WIFI SET WIFI_KEY = '" + String(wifiKey) +
"' WHERE ID = 1;";
     return db_exec(db, sql.c_str());
}

/**
 * Sets the wifi SSID into the SQLite database.
 */
int db_setWifiSSID(sqlite3 *db, String wifiSSID) {
     String sql = "UPDATE WIFI SET WIFI_SSID = '" + String(wifiSSID) +
"' WHERE ID = 1;";
     return db_exec(db, sql.c_str());
}

/**
 * Gets the wifi key from the SQLite database.
 */
String db_getWifiKey(sqlite3 *db) {
     db_response = "";
     db_exec(db, "SELECT WIFI_KEY FROM WIFI WHERE ID = 1;");
     db_response.remove(db_response.length()-1);
     return db_response;
}

/**
 * Gets the wifi ssid from the SQLite database.
 */
String db_getWifiSSID(sqlite3 *db) {
     db_response = "";
     db_exec(db, "SELECT WIFI_SSID FROM WIFI WHERE ID = 1;");
     db_response.remove(db_response.length()-1);
     return db_response;
}
```

```
/*****************************************************/
/*                     ROUTING                       */
/*****************************************************/

/**
 * Inserts the hash of a packet in the routing table.
 */
void insertPacketIntoRecents(String hash) {
     if(routingTable.last == ROUTING_TABLE_SIZE) routingTable.last =
0;
     routingTable.table[routingTable.last] = hash;
     routingTable.last++;
}

/**
 * Checks if the hash of a packet is in the routing table.
 * Returns: 1 if yes, else 0.
 */
int isPacketInRecents(String hash){
     for(int i=0; i<ROUTING_TABLE_SIZE; i++)
       if(routingTable.table[i] == hash) return 1;
     return 0;
}

/**
 * Prints the routing table.
 */
void printRoutingTable(){
     Serial.println("---ROUTING TABLE---");
     for(int i=0; i<ROUTING_TABLE_SIZE; i++) {
       Serial.println("[" + String(i) + "] = " +
routingTable.table[i]);
     }
}



/*****************************************************/
/*                   MESSAGE BUFFER                  */
/*****************************************************/

/**
 * Inserts a message in the buffer.
 */
void insertMessageInBuffer(String msg) {
     if(messageBuffer.n < RECEIVED_BUFFER_SIZE) {
       messageBuffer.buf[messageBuffer.n] = msg;
       messageBuffer.n++;
     }
}

/**
 * Returns number of buffered messages.
 */
int bufferedMessages() {
```

```
     return messageBuffer.n;
}

/**
 * "Frees" the message buffer.
 */
void freeMessageBuffer() {
     messageBuffer.n = 0;
}

/**
 * Prints the message buffer
 */
void printMessageBuffer(){
     Serial.println("---MSG BUFFER---");
     for(int i=0; i<RECEIVED_BUFFER_SIZE; i++) {
       Serial.println("[" + String(i) + "] = " +
messageBuffer.buf[i]);
     }
}




/*********************************************************/
/*                     Cryptography                      */
/*********************************************************/

/**
 * Returns the SHA256 hash of a string.
 */
String strToSHA256(String input) {

     char payload[input.length()+1];
     input.toCharArray(payload, input.length()+1);

     byte shaResult[32];

     mbedtls_md_context_t ctx;
     mbedtls_md_type_t md_type = MBEDTLS_MD_SHA256;
     mbedtls_md_init(&ctx);
     mbedtls_md_setup(&ctx, mbedtls_md_info_from_type(md_type), 0);
     mbedtls_md_starts(&ctx);
     mbedtls_md_update(&ctx, (const unsigned char *) payload,
strlen(payload));
     mbedtls_md_finish(&ctx, shaResult);
     mbedtls_md_free(&ctx);

     String output;
     for(int i= 0; i< sizeof(shaResult); i++){
         char str[3];
         sprintf(str, "%02x", (int)shaResult[i]);
         output += str;
     }
     return output;
}

/**
 * Encrypts a string with AES256.
 * Receives: (key, string)
```

```
 * Returns: encrypted string
 */
String str_AES256_enc(String i_key, String salt, String input) {

      char s2[input.length()+1];
      input.toCharArray(s2, input.length()+1);
      size_t bs = ((input.length()/16)+1)*16;

      char plaintext[bs];
      char encrypted[bs];

      char key[256];
      String o_key = strToSHA256(i_key);
      o_key.toCharArray(key, 256);

      char iv[16];
      String o_iv = strToSHA256(salt).substring(48,64);
      o_iv.toCharArray(iv, 16);

      memset(plaintext, 0, sizeof(plaintext));
      memset(encrypted, 0, sizeof(encrypted));
      strcpy(plaintext, (const char*)s2);

      esp_aes_context aes;
      esp_aes_init(&aes);
      esp_aes_setkey(&aes, (const unsigned char *)key, 256);
      esp_aes_crypt_cbc(&aes, ESP_AES_ENCRYPT, sizeof(plaintext),
(unsigned char*)iv, (uint8_t*)plaintext, (uint8_t*)encrypted);
      esp_aes_free(&aes);
      memset(plaintext, 0, sizeof( plaintext));
      memset(iv, 0, sizeof(iv));

      return encrypted;
}

/**
 * Decrypts an AES256 encrypted string.
 * Receives: (key, string)
 * Returns: plaintext string
 */
String str_AES256_dec(String i_key, String salt, String input) {

      char s2[input.length()+1];
      input.toCharArray(s2, input.length()+1);
      size_t bs = ((input.length()/16)+1)*16;

      char plaintext[bs];
      char encrypted[bs];

      char key[256];
      String o_key = strToSHA256(i_key);
      o_key.toCharArray(key, 256);

      char iv[16];
      String o_iv = strToSHA256(salt).substring(48,64);
      o_iv.toCharArray(iv, 16);

      memset(encrypted, 0, sizeof(encrypted));
      strcpy(encrypted, (const char*)s2);

      esp_aes_context aes;
```

```
    esp_aes_init(&aes);
    esp_aes_setkey(&aes, (const unsigned char *)key, 256);
    esp_aes_crypt_cbc(&aes, ESP_AES_DECRYPT, sizeof(encrypted),
(unsigned char*)iv, (uint8_t*)encrypted, (uint8_t*)plaintext);
    esp_aes_free(&aes);

    return plaintext;
}
```

16.2 Código HTML

```
<!DOCTYPE html>
<html>

  <title>LoRaChat</title>

  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1,
user-scalable=0">

  <link rel="stylesheet" href="lorachat.css">
  <script src="lorachat.js"></script>

  <body class="w3-black" onload="gen_onPageLoad()">

    <!-- Sidebar -->
    <div id="sidebar" class="w3-sidebar w3-border-right border-lorachat w3-bar-block w3-
collapse w3-card w3-animate-left w3-black" style="width:350px;">
      <button class="w3-bar-item w3-blue w3-button w3-large w3-hide-large w3-right"
onclick="gui_sidebarClose()">&times;</button>
      <div class="w3-container w3-blue" style="font-weight: lighter;">
        <h4>LoRaChat</h4>
      </div>
      <!-- Here sidebar items -->
    </div>

    <!-- Page Content -->
    <div class="w3-main" style="margin-left:350px;">

      <!-- Title bar -->
      <div class="w3-bar w3-blue" style="height:50px;">
          <span class="w3-bar-item w3-button w3-cell w3-blue w3-padding w3-hover-blue
w3-xlarge w3-hide-large" onclick="gui_sidebarOpen()">&#9776;</span>
          <span id="chat-title" class="w3-bar-item w3-xlarge w3-padding w3-blue"></span>
          <div class="w3-dropdown-click w3-right">
              <span onclick="gui_dropdownMenuChangeStatus()" class="w3-button w3-blue
w3-hover-blue w3-xlarge">&#10998;</span>
              <div id="dropdown-menu" class="w3-dropdown-content w3-bar-block"
style="right:0">
                <a class="w3-bar-item w3-button w3-dark-gray w3-hover-grey"
onclick="gui_openAddChatModal()">Add chat</a>
                <a class="w3-bar-item w3-button w3-dark-gray w3-hover-grey"
onclick="gui_openContactsModal()">Contacts</a>
                <a class="w3-bar-item w3-button w3-dark-gray w3-hover-grey"
onclick="gui_openMainSettingsModal()">Main settings</a>
                <a class="w3-bar-item w3-button w3-dark-gray w3-hover-grey"
onclick="gui_saveDatabase()">Save database</a>
              </div>
          </div>
      </div>
```

```
      <!-- Chat messages screen -->
      <div id="chat-msg-screen" class="w3-container w3-black" style="height:85vh;
overflow-y: scroll; padding-top: 10px;"></div>

      <!-- Typing box and sending button -->
      <footer id="chat-box" class="w3-bar w3-black" style="position: sticky; height:5%;
padding:5px; bottom:0;">
        <input id="chat-input-box" type="text" class="w3-bar-item chat-input w3-round-
xxlarge w3-border" placeholder="Type a message..." style="height:45px; width:85%;"
onkeypress="gui_sendMsgOnEnter(event)">
        <button class="w3-bar-item w3-button w3-blue w3-round-xxlarge w3-right hover-
light-blue" style="height:45px; width:14%; text-align: center;"
onclick="gui_sendMsg()">&#10148;</button>
      </footer>

      <!-- MODALS -->
      <!-- Main settings modal -->
      <div id="main-settings-modal" class="w3-modal">
        <div class="w3-modal-content w3-black w3-border">

          <header class="w3-container w3-blue">
            <span onclick="gui_clearMainSettingsModal()"
            class="w3-button w3-display-topright">&times;</span>
            <h2>Main Settings</h2>
          </header>

          <div class="w3-container">
            <p>
              <label class="w3-text-blue"><b>User ID</b></label><br>
              <input id="main-settings-userid-input" class="w3-input w3-dark-grey w3-
border w3-round-xxlarge" type="text">
            </p>
            <p>
              <button class="w3-btn w3-blue w3-round-xxlarge w3-hover-light-blue"
onclick="gui_changeUserId()">Change</button>
            </p>
            <p>
              <label class="w3-text-blue"><b>Username</b></label><br>
              <input id="main-settings-username-input" class="w3-input w3-dark-grey w3-
border w3-round-xxlarge" type="text">
            </p>
            <p>
              <button class="w3-btn w3-blue w3-round-xxlarge w3-hover-light-blue"
onclick="gui_changeUserName()">Change</button>
            </p>
            <p>
              <label class="w3-text-blue"><b>Wifi SSID</b></label><br>
              <input id="main-settings-ssid-input" class="w3-input w3-dark-grey w3-
border w3-round-xxlarge" type="text">
            </p>
```

```
            <p>
              <button class="w3-btn w3-blue w3-round-xxlarge w3-hover-light-blue"
onclick="gui_changeWifiSSID()">Change</button>
            </p>
            <p>
              <label class="w3-text-blue"><b>Wifi key</b></label><br>
              <input id="main-settings-key-input" class="w3-input w3-dark-grey w3-border
w3-round-xxlarge" type="password">
            </p>
            <p>
              <button class="w3-btn w3-blue w3-round-xxlarge w3-hover-light-blue"
onclick="gui_changeWifiKey()">Change</button>
            </p>
            <p>
              <button class="w3-btn w3-red w3-round-xxlarge w3-hover-light-grey"
onclick="gui_clearMainSettingsModal()">Cancel</button>
            </p>
          </div>

        </div>
      </div>

      <!-- Conversation settings modal -->
      <div id="conv-settings-modal" class="w3-modal">
        <div class="w3-modal-content w3-black w3-border">

          <header class="w3-container w3-blue">
            <span onclick="gui_clearChatSettingsModal()"
            class="w3-button w3-display-topright">&times;</span>
            <h2>Conversation settings</h2>
          </header>

          <div class="w3-container">
            <p>
              <label class="w3-text-blue"><b>Chat title</b></label><br>
              <input id="chat-settings-change-title" class="w3-input w3-dark-grey w3-
border w3-round-xxlarge" type="text" maxlength="30">
            </p>
            <p>
              <button class="w3-btn w3-blue w3-round-xxlarge w3-hover-light-blue"
onclick="gui_changeChatTitle()">Change</button>
            </p>
            <p>
              <label class="w3-text-blue"><b>Chat encryption key</b></label><br>
              <input id="chat-settings-change-key" class="w3-input w3-dark-grey w3-
border w3-round-xxlarge" type="text" maxlength="16">
            </p>
            <p>
              <button class="w3-btn w3-blue w3-round-xxlarge w3-hover-light-blue"
onclick="gui_changeChatKey()">Change</button>
            </p>
```

```
            <p id="chat-settings-id-label">Chat ID:</p>
            <p>
              <button class="w3-btn w3-yellow w3-round-xxlarge w3-hover-orange"
onclick="gui_removeChat()">Remove chat</button>
              <button class="w3-btn w3-red w3-round-xxlarge w3-hover-light-grey"
onclick="gui_clearChatSettingsModal()">Cancel</button>
            </p>
          </div>
        </div>
      </div>

      <!-- Add chat modal -->
      <div id="add-chat-modal" class="w3-modal">
        <div class="w3-modal-content w3-black w3-border">

          <header class="w3-container w3-blue">
            <span onclick="gui_clearAddChatModal()"
            class="w3-button w3-display-topright">&times;</span>
            <h2>Add chat</h2>
          </header>

          <div class="w3-container">
              <p>
                <label class="w3-text-blue"><b>Chat title</b></label><br>
                <input id="add-chat-input-title" class="w3-input w3-dark-grey w3-border
w3-round-xxlarge" type="text" maxlength="30">
              </p>
              <p>
                <label class="w3-text-blue"><b>Chat ID</b></label><br>
                <input id="add-chat-input-id" class="w3-input w3-dark-grey w3-border w3-
round-xxlarge" type="text">
              </p>
              <p>
                <label class="w3-text-blue"><b>Chat encryption key</b></label><br>
                <input id="add-chat-input-key" class="w3-input w3-dark-grey w3-border
w3-round-xxlarge" type="text" maxlength="50">
              </p>
              <p>
                <button class="w3-btn w3-red w3-round-xxlarge w3-hover-light-grey"
onclick="gui_clearAddChatModal()">Cancel</button>
                <button class="w3-btn w3-blue w3-round-xxlarge w3-hover-light-blue"
onclick="gui_addChat()">Add</button>
              </p>
          </div>
        </div>
      </div>

        <!-- Contacts modal -->
      <div id="contacts-modal" class="w3-modal">
        <div class="w3-modal-content w3-black w3-border">
```

```
          <header class="w3-container w3-blue">
            <span onclick="gui_clearContactsModal()"
            class="w3-button w3-display-topright">&times;</span>
            <h2>Contacts</h2>
          </header>

          <div class="w3-container">
              <p>
                <label class="w3-text-blue"><b>Contact list</b></label><br>
              </p>
              <div class="w3-container w3-black" style="height:30vh; overflow-
y:scroll;">
                <ul id="contact-list" class="w3-ul w3-dark-grey" style="overflow-
y:scroll;"></ul>
              </div>
              <p>
                <button class="w3-btn w3-blue w3-round-xxlarge w3-hover-light-blue"
onclick="gui_openAddOrChangeContactModal()">Add contact</button>
              </p>
          </div>
        </div>
      </div>

        <!-- Add or change contact modal -->
      <div id="add-change-contact-modal" class="w3-modal">
        <div class="w3-modal-content w3-black w3-border">

          <header class="w3-container w3-blue">
            <span onclick="gui_clearAddOrChangeContactModal()"
            class="w3-button w3-display-topright">&times;</span>
            <h2 id="add-change-contact-title">Add or change Contact</h2>
          </header>

          <div class="w3-container">
              <p>
                <label class="w3-text-blue"><b>Contact name</b></label><br>
                <input id="add-change-contact-input-name" class="w3-input w3-dark-grey
w3-border w3-round-xxlarge" type="text" maxlength="15">
              </p>
              <p id="add-change-contact-modal-contact-id">
                <label id="add-change-contact-modal-contact-id-label" class="w3-text-
blue"><b>Contact ID</b></label><br>
              </p>
              <p>
                <button class="w3-btn w3-red w3-round-xxlarge w3-hover-light-grey"
onclick="gui_clearAddOrChangeContactModal()">Cancel</button>
                <button id="add-change-contact-button" class="w3-btn w3-blue w3-round-
xxlarge w3-hover-light-blue"></button>
              </p>
          </div>
        </div>
```

```
      </div>

      <!-- Alert modal -->
      <div id="alert-modal" class="w3-modal">
          <div class="w3-modal-content">
              <div id="alert-panel">
                <span onclick="document.getElementById('alert-
modal').style.display='none';" class="w3-button w3-large w3-display-
topright">&times;</span>
                <p id="alert-text"></p>
              </div>
          </div>
      </div>

      <!-- WebSocket modal -->
      <div id="websocket-modal" class="w3-modal">
          <div class="w3-modal-content w3-yellow w3-padding">
              <div>
                <p> WebSocket disconnected. Waiting....</p>
              </div>
          </div>
      </div>

    </div>
  </body>
</html>
```

16.3 Código Javascript

```
/**
 *
 * LoRaChat Javascript.
 * Implements the engine of the LoRaChat web client.
 *
 */


/*** LORACHAT CODES ***/
var LORA_MSG = 0;
var LORA_ACK = 1;
/**********************/

/*** WEBSOCKET CODES ***/
var WS_ERROR = -1;
var WS_SEND_MSG = 0;
var WS_RECV_MSG = 1;
var WS_RECV_ACK = 2;
var WS_MSG_SENT = 3;
var WS_DB_REQ = 4;
var WS_DB_SEND = 5;
var WS_DB_RECV = 6;
var WS_ADD_CHAT = 7;
var WS_DEL_CHAT = 8;
var WS_SET_CHAT_KEY = 9;
var WS_SET_WIFI_SSID = 10;
var WS_SET_WIFI_KEY = 11;
var WS_BUFFERED_MSG = 12;
/**********************/

/*** MESSAGE STATUS CODES ***/
var MSG_PENDING = 0;
var MSG_SENT = 1;
var MSG_RECEIVED = 2;
/****************************/

var WS; // WebSocket
var DATABASE;
var ACTIVE_CHAT;
var SOCKET_READY = false;
var DATABASE_LOADED = false;


/**********************/
/*      GENERAL       */
/**********************/

/**
 * Performed before closing the window.
```

```
 */
window.onbeforeunload = function() {
    // Save (send) the database
    com_sendPacket(com_generateSendDatabasePacket());
};

/**
 * Asks for notification permission.
 */
function gen_askForNotificationPermission() {
    if (!("Notification" in window)) return;
    if (Notification.permission !== "granted")
        Notification.requestPermission();
}

/**
 * Notifies a new message in a chat.
 * @param {string} chat
 */
function gen_notify(chat) {
    if (!("Notification" in window)) return;
    if (Notification.permission === "granted")
        new Notification("New message on " + chat);
}

/**
 * Initializes the WebSocket.
 */
function gen_initWebSocket() {

    // Connect to WebSocket server
    WS = new WebSocket("ws://192.168.4.1:81");

    WS.onopen = function() {
        SOCKET_READY = true;
        if(!DATABASE_LOADED) {
            com_requestDatabase();
        }
        com_sendPacket(com_generateBufferedMsgRequestPacket());
        gui_closeWebSocketModal();
    }

    WS.onmessage = function(evt) {
        com_packetHandling(evt.data);
    };

    WS.onclose = function() {
        SOCKET_READY = false;
        gui_openWebSocketModal();
        WS = null;
        setTimeout(gen_initWebSocket, 1000);
```

```
    }

    WS.onerror = function() {
        gui_alert("WebSocket error.", "red");
    }
}

/**
 * Waits until websocket is connected.
 * @param {WebSocket} socket
 */
function gen_waitForSocketConnection(){
    setTimeout(
        function () {
            if (!SOCKET_READY) {
                gen_waitForSocketConnection();
            }
        }, 5);
}

/**
 * Executed when the page is loaded.
 */
function gen_onPageLoad(){
    gui_openWebSocketModal();
    gen_initWebSocket();
    gui_loadGUI(DATABASE);
    document.getElementById("chat-msg-screen").scrollTo(0,
document.getElementById("chat-msg-screen").scrollHeight);
    gen_askForNotificationPermission();
}

/**
 * Generates and returns a complex message id:
 * chat-author-msg
 * @param {int} chatId
 * @param {int} author
 * @param {int} msgId
 * @returns chat-author-msg
 */
function gen_generateMsgId(chatId, author, msgId) {
    return chatId + "-" + author + "-" + msgId;
}

/**
 * Parses a complex msgId and returns a list:
 * [0] = chat ID
 * [1] = author
 * [2] = message ID
 * @param {string} msgId message id (chat-author-msg)
 * @returns [chatId, author, messageId]
```

```
 */
function gen_parseMsgId(msgId) {
    return msdId.split("-");
}



/*************************************/
/*      Communication with ESP32     */
/*************************************/

/**
 * Handles with websocket received messages.
 * @param {string} p received data
 */
function com_packetHandling(p) {
    var msg = JSON.parse(p);
    switch (msg["type"]) {
        case WS_RECV_MSG: com_receivedNewMessageHandler(msg); break;
        case WS_RECV_ACK: com_receivedACKHandler(msg); break;
        case WS_MSG_SENT: com_msgSentConfirmationHandler(msg); break;
        case WS_DB_SEND: com_receivedDBHandler(msg); break;
        case WS_DB_RECV: com_receivedDBConfirmationHandler(); break;
        default: break;
    }
}

/**
 * Sends a packet (string) trough the web socket.
 * @param {string} p packet to send
 */
function com_sendPacket(p){
    try {
        WS.send(p);
    } catch (error) {
        gui_alert("WebSocket error: " + error, "red");
    }
}

/**
 * Handles with received text messages: saves the message
 * into the database hand does GUI stuff.
 * @param {string} msg received message
 */
function com_receivedNewMessageHandler(msg) {
    var msgId = gen_generateMsgId(msg["chatId"], msg["author"], msg["msgId"]);
    db_addRemoteMsgToChat(DATABASE, msg["chatId"], msg["msgId"],
msg["author"], msg["text"]);
    if(ACTIVE_CHAT == msg["chatId"])
        gui_addReceivedMsg(msgId, msg["author"], msg["text"]);
    else {
```

```
        gui_addLastMsgToChat(msg["chatId"], msg["author"], msg["text"]);
        gui_removeBubbleFromChat(msg["chatId"]);
        var unread = db_getChatUnreadCounter(DATABASE, msg["chatId"]);
        db_setChatUnreadCounter(DATABASE, msg["chatId"], unread+1);
        gui_addBubbleToChat(msg["chatId"], unread+1);
        gui_moveChatListItemFirst(msg["chatId"]);
        gui_moveChatListItemFirst(ACTIVE_CHAT);
        gen_notify(db_getChatTitle(DATABASE, msg["chatId"]));
    }
}

/**
 * Handles with received ACK messages: sets the message as
 * read in both database and GUI.
 * @param {JSON} msg ack message
 */
function com_receivedACKHandler(msg) {
    var msgId = gen_generateMsgId(msg["chatId"], msg["author"], msg["msgId"]);
    db_setMessageStatus(DATABASE, msg["chatId"], msgId, MSG_RECEIVED); // Set
on database
    gui_setMsgStatus(msg["chatId"], msgId, MSG_RECEIVED); // Set on screen
}

/**
 * Handles with message sent confirmations: sets the message
 * status as sent int he database and refreshes the status on
 * the GUI if the message's chat is the active one.
 * @param {JSON} msg
 */
function com_msgSentConfirmationHandler(msg) {
    var msgId = gen_generateMsgId(msg["chatId"], msg["author"], msg["msgId"]);
    db_setMessageStatus(DATABASE, msg["chatId"], msgId, MSG_SENT);
    gui_setMsgStatus(msg["chatId"], msgId, MSG_SENT);
}

/**
 * Handles with the received database.
 * @param {JSON} msg
 */
function com_receivedDBHandler(msg) {
    DATABASE = msg["db"];
    DATABASE_LOADED = true;
    gui_loadGUI(DATABASE);
}

/**
 * Handles with received database confirmation (when the
 * server has correctly received the database from the
 * client): shows a notification.
 */
function com_receivedDBConfirmationHandler() {
```

```
    gui_alert("Database saved!", "green");
}

/**
 * Generates a packet to send a lora text message trough websocket.
 * @param {int} chatId chat id
 * @param {int} msgId message id
 * @param {string} text message text
 */
function com_generateSendMessagePacket(chatId, msgId, text) {
    return WS_SEND_MSG + "|" + chatId + "|" + msgId + "|" +
db_getUserId(DATABASE) + "|" + text;
}

/**
 * Generates a packet to send the database trough websocket.
 */
function com_generateSendDatabasePacket() {
    return WS_DB_SEND + "|" + JSON.stringify(DATABASE);
}

/**
 * Generates a database request websocket packet.
 */
function com_generateDatabaseRequestPacket() {
    return WS_DB_REQ + "|";
}

/**
 * Generates a buffered messages request websocket packet.
 */
function com_generateBufferedMsgRequestPacket() {
    return WS_BUFFERED_MSG + "|";
}

/**
 * Generates an add chat websocket packet.
 * @param {int} chatId
 * @param {string} key
 */
function com_generateAddChatPacket(chatId, key){
    return WS_ADD_CHAT + "|" + chatId + "|" + key;
}

/**
 * Generates a delete chat websocket packet.
 * @param {int} chatId
 */
function com_generateDelChatPacket(chatId) {
    return WS_DEL_CHAT + "|" + chatId;
}
```

```
function com_generateSetChatKeyPacket(chatId, key) {
    return WS_SET_CHAT_KEY + "|" + chatId + "|" + key;
}

function com_generateSetWifiSSIDPacket(ssid) {
    return WS_SET_WIFI_SSID + "|" + ssid;
}

function com_generateSetWifiKeyPacket(key) {
    return WS_SET_WIFI_KEY + "|" + key;
}

/**
 * Requests the database over the websocket.
 */
function com_requestDatabase() {
    com_sendPacket(com_generateDatabaseRequestPacket());
}


/*********************************/
/*      Database functions       */
/*********************************/

/**
 * Returns and empty JSON database.
 */
function db_createDB(){
    return {
        "user_id": undefined,
        "username": undefined,
        "wifi_ssid": "LoRaChat",
        "wifi_key": "",
        "contacts": [],
        "chats": []
    };
}

/**
 * Returns a contact in JSON format
 * @param {int} userId contact id
 * @param {string} name contact name
 */
function db_createContact(userId, name){
    return {
        "id": userId,
        "name": name
    }
}
```

```
/**
 * Returns a chat in JSON format.
 * @param {int} chatId chat id
 * @param {string} title chat title
 * @param {string} key chat key
 */
function db_createChat(chatId, title, key){
    return {
        "id": chatId,
        "title": title,
        "key": key,
        "unread": 0,
        "id_counter": 0,
        "messages": []
    }
}

/**
 * Returns a message in JSON format.
 * @param {string} id message id (chat-author-msg)
 * @param {string} author author of message
 * @param {string} text text of message
 * @param {boolean} mine true if it is mine, false instead
 */
function db_createMsg(id, author, text, mine){
    return {
        "id": id,
        "mine": mine, // Is the message mine?
        "author": author,
        "text": text,
        "status": MSG_PENDING // false = sent, true = received.
    }
}

/**
 * Adds a contact to the database.
 * @param {json} db database
 * @param {int} userId contact id
 * @param {string} name contact name
 */
function db_addContact(db, userId, name){
    db.contacts.push(db_createContact(userId, name));
}

/**
 * Adds a chat to local and server's database.
 * Also updates user's database on server side.
 * @param {json} db database
 * @param {int} chatId chat id
 * @param {string} title chat title
 * @param {string} key chat key
```

```
 */
function db_addChat(db, chatId, title, key){
    db.chats.push(db_createChat(chatId, title, key));
    com_sendPacket(com_generateAddChatPacket(chatId,key));
    com_sendPacket(com_generateSendDatabasePacket()); // Send all database
}

/**
 * Adds a remote message to a chat in the database.
 * @param {json} db database
 * @param {int} chatId chat id
 * @param {int} msgId message id (simple)
 * @param {string} author sender
 * @param {string} text text of message
 */
function db_addRemoteMsgToChat(db, chatId, msgId, author, text){
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId)
            db.chats[i].messages.push(db_createMsg(gen_generateMsgId(chatId,
author, msgId), author, text, false));
}

/**
 * Adds an own message to a chat in the database.
 * @param {json} db database
 * @param {int} chatId chat id
 * @param {int} msgId message id (simple)
 * @param {string} text text of message
 */
function db_addOwnMsgToChat(db, chatId, msgId, text){
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId)
            db.chats[i].messages.push(db_createMsg(gen_generateMsgId(chatId,
db.user_id, msgId), db.user_id, text, true));
}

/**
 * Sets the name of a contact in the database.
 * @param {json} db database
 * @param {int} contactId contact id
 * @param {string} name contact name
 */
function db_setContactName(db, contactId, name) {
    for (var i = 0; i < db.contacts.length; i++)
        if(db.contacts[i].id === contactId) {
            db.contacts[i].name = name;
            return;
        }
}

/**
```

```
 * Sets the contact ID of a contact in the database.
 * @param {json} db database
 * @param {int} oldId old id
 * @param {int} newId new id
 */
function db_setContactId(db, oldId, newId) {
    for (var i = 0; i < db.contacts.length; i++)
        if(db.contacts[i].id === oldId) {
            db.contacts[i].id = newId;
            return;
        }
}

/**
 * Returns the name of a contact in the database.
 * If the contact is not saved, returns the id.
 * @param {json} db database
 * @param {int} contactId contact id
 */
function db_getContactName(db, contactId) {
    for (var i = 0; i < db.contacts.length; i++)
        if(db.contacts[i].id === contactId)
            return db.contacts[i].name;
    return contactId;
}

/**
 * Returns true if the contact exists in the database.
 * Returns false if the contact does not exist.
 * @param {json} db database
 * @param {int} contactId contact id
 */
function db_existsContact(db, contactId) {
    for (var i = 0; i < db.contacts.length; i++)
        if(db.contacts[i].id === contactId)
            return true;
    return false;
}

function db_removeContact(db, contactId) {
    for (var i = 0; i < db.contacts.length; i++)
        if(db.contacts[i].id === contactId)
            db.contacts.splice(i,1);
}

/**
 * Returns a chat from the database in JSON format.
 * @param {json} db database
 * @param {int} chatId chat id
 */
function db_getChat(db, chatId) {
```

```
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId)
            return db.chats[i];
}

/**
 * Removes a chat from the local and server's database.
 * Also updates user's database on server side.
 * @param {json} db database
 * @param {int} chatId chat id
 */
function db_removeChat(db, chatId) {
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId) {
            db.chats.splice(i,1);
            com_sendPacket(com_generateDelChatPacket(chatId));
            com_sendPacket(com_generateSendDatabasePacket());
            return;
        }
}

/**
 * Returns all messages from a chat in JSON format.
 * @param {json} db database
 * @param {int} chatId chat id
 */
function db_getChatMsgs(db, chatId) {
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId)
            return db.chats[i].messages;
}

/**
 * Sets user id in the database
 * @param {json} db database
 * @param {string} name user id
 */
function db_setUserId(db, userId) {
    db.user_id = userId;
}

/**
 * Returns the user id from the database.
 * @param {json} db database
 */
function db_getUserId(db) {
    return db.user_id;
}

/**
 * Sets username in the database.
```

```
 * @param {json} db database
 * @param {string} username username
 */
function db_setUsername(db, username) {
    db.username = username;
}

/**
 * Returns username from the database.
 * @param {json} db database
 */
function db_getUsername(db) {
    return db.username;
}

/**
 * Sets the wifi SSID in the database.
 * Also updates user's database on server side.
 * @param {json} db database
 * @param {string} ssid wifi SSID
 */
function db_setWifiSSID(db, ssid){
    com_sendPacket(com_generateSetWifiSSIDPacket(ssid));
    db.wifi_ssid = ssid;
    com_sendPacket(com_generateSendDatabasePacket());
}

/**
 * Returns the wifi SSID from the database.
 * @param {json} db database
 */
function db_getWifiSSID(db){
    return db.wifi_ssid;
}

/**
 * Sets the wifi key in the local and server's database.
 * Also updates user's database on server side.
 * @param {json} db database
 * @param {string} key wifi key
 */
function db_setWifiKey(db, key){
    db.wifi_key = key;
    com_sendPacket(com_generateSetWifiKeyPacket(key));
    com_sendPacket(com_generateSendDatabasePacket());
}

/**
 * Returns the wifi key from the database
 * @param {json} db
 */
```

```
function db_getWifiKey(db){
    return db.wifi_key;
}

/**
 * Sets the title of a chat in the database.
 * @param {json} db database
 * @param {int} chatId chat id
 * @param {string} title chat title
 */
function db_setChatTitle(db, chatId, title){
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId) {
            db.chats[i].title = title;
            return;
        }
}

/**
 * Returns the title of a chat in the database.
 * @param {json} db
 * @param {int} chatId
 */
function db_getChatTitle(db, chatId){
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId)
            return db.chats[i].title;
}

/**
 * Sets the key of a chat in the local and server's database.
 * Also updates user's database on server side.
 * @param {json} db database
 * @param {int} chatId chat id
 * @param {string} key key
 */
function db_setChatKey(db, chatId, key){
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId) {
            db.chats[i].key = key;
            com_sendPacket(com_generateSetChatKeyPacket(chatId, key));
            com_sendPacket(com_generateSendDatabasePacket());
            return;
        }
}

/**
 * Returns the key of a chat in the database.
 * @param {json} db database
 * @param {int} chatId chat id
 */
```

```
function db_getChatKey(db, chatId){
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId)
            return db.chats[i].key;
}

/**
 * Sets the unread counter of a chat in the database.
 * @param {json} db database
 * @param {int} chatId chat id
 * @param {int} n counter value
 */
function db_setChatUnreadCounter(db, chatId, n) {
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId) {
            db.chats[i].unread = n;
            return;
        }
}

/**
 * Returns the unread messages counter from a chat in the database.
 * @param {json} db  database
 * @param {int} chatId chat id
 */
function db_getChatUnreadCounter(db, chatId) {
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId)
            return db.chats[i].unread;
    return 0;
}

/**
 * Sets the status of a message in the database.
 * @param {json} db database
 * @param {int} chatId chat id
 * @param {string} msgId message id (chat-author-msg)
 * @param {int} status status
 */
function db_setMessageStatus(db, chatId, msgId, status) {
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id == chatId) {
            var chat = db.chats[i];
            for (var j = 0; j < chat.messages.length; j++)
                if(chat.messages[j].id == msgId) {
                    chat.messages[j].status = status;
                    return;
                }
        }
}
```

```
/**
 * Returns the message id counter from a chat in the database.
 * @param {json} db database
 * @param {int} chatId chat id
 * @param {int} n number of the counter
 */
function db_setChatIdCounter(db, chatId, n) {
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId) {
            db.chats[i].id_counter = n;
            return;
        }
}

/**
 * Returns the id counter from a chat in the database.
 * @param {json} db database
 * @param {int} chatId chat id
 */
function db_getChatIdCounter(db, chatId) {
    for (var i = 0; i < db.chats.length; i++)
        if(db.chats[i].id === chatId)
            return db.chats[i].id_counter;
}




/************************************/
/*      GUI handling functions      */
/************************************/

/**
 * Loads the GUI with the data of the database.
 * Only loads the sidebar and cleans the screen.
 * @param {json} db database
 */
function gui_loadGUI(db){
    gui_setChatScreenTitle("");
    gui_cleanChatInputBox();
    gui_hideChatBox();
    if(db === undefined) return;
    for (var i = 0; i < db.chats.length; i++) {
        var chat = db.chats[i];
        if(chat.messages[chat.messages.length-1] === undefined) {
            var lastMsg = undefined;
            var lastAuthor = undefined;
        } else{
            var lastMsg = chat.messages[chat.messages.length-1].text;
            var lastAuthor = db_getContactName(DATABASE,
chat.messages[chat.messages.length-1].author);
```

```
        }
        gui_addChatToSidebar2(
            chat.id,
            chat.title,
            lastAuthor,
            lastMsg,
            chat.unread);
        if(chat.unread>0) { // If there are unread messages
            gui_moveChatListItemFirst(chat.id); // Move first place
        }
    }
}

/**
 * Generates an alert in the gui.
 * @param {string} text message to show
 * @param {string} color color of the alert
 */
function gui_alert(text, color){
    var alert = document.getElementById('alert-panel');
    alert.className = "w3-panel w3-display-container";
    switch (color) {
        case "red": alert.className += " w3-red"; break;
        case "green": alert.className += " w3-green"; break;
        case "yellow": alert.className += " w3-yellow"; break;
        default: alert.className += " w3-blue"; break;
    }
    var alertText = document.getElementById('alert-text');
    alertText.innerHTML = text;
    document.getElementById('alert-modal').style.display="block";
}

/**
 * Opens sidebar.
 */
function gui_sidebarOpen() {
    document.getElementById("sidebar").style.display = "block";
}

/**
 * Hides sidebar.
 */
function gui_sidebarClose() {
    document.getElementById("sidebar").style.display = "none";
}

/**
 * Shows chat box.
 */
function gui_showChatBox() {
    document.getElementById("chat-box").style.display = "block";
```

```
}

/**
 * Hides chat box.
 */
function gui_hideChatBox() {
    document.getElementById("chat-box").style.display = "none";
}

/**
 * Cleans the chat screen.
 */
function gui_clearChatScreen() {
    var screen = document.getElementById("chat-msg-screen");
    screen.innerHTML = '';
}

/**
 * Sets the chat screen title.
 * @param {string} title title to set
 */
function gui_setChatScreenTitle(title) {
    document.getElementById("chat-title").innerHTML = title;
}

/**
 * Adds active chat item to the dropdown menu.
 */
function gui_addActiveChatMenuItem() {
    var item = document.createElement("a");
    item.id = "dropdown-active-chat-item";
    item.className = "w3-bar-item w3-button w3-dark-gray w3-hover-grey";
    item.innerHTML = "Current chat settings";
    item.addEventListener("click", function() { // Click listener
        gui_openChatSettingsModal();
    });
    var parent = document.getElementById("dropdown-menu");
    parent.insertBefore(item, parent.childNodes[3]);
}

/**
 * Removes active chat item to the dropdown menu.
 */
function gui_removeActiveChatMenuItem() {
    var elem = document.getElementById("dropdown-active-chat-item");
    elem.parentNode.removeChild(elem);
}

/**
 * Opens and closes the dropdown menu.
 */
```

```
function gui_dropdownMenuChangeStatus() {
    var x = document.getElementById("dropdown-menu");
    if (x.className.indexOf("w3-show") == -1) x.className += " w3-show";
    else x.className = x.className.replace(" w3-show", "");
}

/**
 * Opens and closes the dropdown menu.
 */
function gui_dropdownMenuClose() {
    document.getElementById("dropdown-menu").className.replace(" w3-show",
"");
}

/**
 * Shows add chat modal.
 */
function gui_openAddChatModal() {
    document.getElementById('add-chat-modal').style.display='block';
    gui_dropdownMenuChangeStatus(); // Close menu
}

/**
 * Shows chat settings modal.
 */
function gui_openChatSettingsModal() {
    if(ACTIVE_CHAT != undefined) { // Show current values
        document.getElementById('chat-settings-change-title').value =
db_getChatTitle(DATABASE, ACTIVE_CHAT);;
        document.getElementById('add-chat-input-id').value = ACTIVE_CHAT;
        document.getElementById('chat-settings-change-key').value =
db_getChatKey(DATABASE, ACTIVE_CHAT);
        document.getElementById('chat-settings-id-label').innerHTML = "Chat
ID: " + ACTIVE_CHAT;
    }
    document.getElementById('conv-settings-modal').style.display='block';
    gui_dropdownMenuChangeStatus(); // Close menu
}

/**
 * Adds a contact to the contacts modal contact list.
 * @param {int} contactId contact id
 * @param {string} contactName contact name
 */
function gui_addContactToContactsModalList(contactId, contactName) {
    var item = document.createElement("li"); // list item
    item.id = "contact-list-" + contactId;
    item.className = "w3-border-black";
    var name = document.createElement("label"); // contact name
    name.innerHTML = contactName + " (ID: " + contactId + ")";
    item.appendChild(name); // append contact name
```

```
    var removeButton = document.createElement("button"); // Remove contact
button
    removeButton.id = "remove-contact-button-" + contactId;
    removeButton.className = "w3-btn w3-right w3-tiny w3-red w3-round-xxlarge
w3-hover-light-grey";
    removeButton.innerHTML = "Remove";
    removeButton.addEventListener("click", function() { // Click listener
        gui_removeContact(contactId);
    });
    item.appendChild(removeButton); // append remove contact button

    var changeButton = document.createElement("button"); // Change contact
button
    changeButton.id = "change-contact-button-" + contactId;
    changeButton.className = "w3-btn w3-right w3-tiny w3-blue w3-round-xxlarge
w3-hover-light-blue";
    changeButton.innerHTML = "Change";
    changeButton.addEventListener("click", function() { // Click listener
        gui_openAddOrChangeContactModal(contactId);
    });
    item.appendChild(changeButton); // append change contact button
    document.getElementById("contact-list").appendChild(item);
}

/**
 * Shows contacts modal.
 */
function gui_openContactsModal() {
    for (var i = 0; i < DATABASE.contacts.length; i++) // Load contacts
        gui_addContactToContactsModalList(DATABASE.contacts[i].id,
DATABASE.contacts[i].name);
    document.getElementById('contacts-modal').style.display='block';
    gui_dropdownMenuChangeStatus(); // Close menu
}

/**
 * Opens add or change contact modal.
 * If the contact ID is defined: change
 * If the contact ID is not defined: add
 * @param {int} contactId contact id
 */
function gui_openAddOrChangeContactModal(contactId) {
    if(contactId == undefined) { // ADD
        document.getElementById('add-change-contact-title').innerHTML = "Add
contact";
        var label = document.getElementById('add-change-contact-modal-contact-
id-label');
        label.className = "w3-text-blue";
        label.innerHTML = "Contact ID";
        var idInput = document.createElement("input");
```

```
        idInput.id = "add-change-contact-input-id";
        idInput.className = "w3-input w3-dark-grey w3-border w3-round-
xxlarge";
        document.getElementById('add-change-contact-modal-contact-
id').appendChild(idInput);
        var addButton = document.getElementById('add-change-contact-button');
        addButton.innerHTML = "Add";
        addButton.addEventListener("click", function() { // Click listener
            gui_addContact();
        });
    } else { // CHANGE
        document.getElementById('add-change-contact-title').innerHTML =
"Change contact";
        var label = document.getElementById('add-change-contact-modal-contact-
id-label');
        label.className = "w3-text-white";
        label.innerHTML = "Contact ID: " + contactId;
        var changeButton = document.getElementById('add-change-contact-
button');
        changeButton.innerHTML = "Change";
        document.getElementById('add-change-contact-input-name').value =
db_getContactName(DATABASE, contactId);
        //document.getElementById('add-change-contact-input-id').value =
contactId;
        changeButton.addEventListener("click", function() { // Click listener
            gui_changeContact(contactId);
        });
    }
    document.getElementById('add-change-contact-modal').style.display='block';
}

/**
 * Shows main settings modal.
 */
function gui_openMainSettingsModal() {
    var userId = db_getUserId(DATABASE);
    if(userId !== undefined) document.getElementById('main-settings-userid-
input').value = db_getUserId(DATABASE);
    else document.getElementById('main-settings-userid-input').value = "";
    var username = db_getUsername(DATABASE);
    if(username !== undefined) document.getElementById('main-settings-
username-input').value = db_getUsername(DATABASE);
    else document.getElementById('main-settings-username-input').value = "";
    document.getElementById('main-settings-ssid-input').value =
db_getWifiSSID(DATABASE);
    document.getElementById('main-settings-key-input').value =
db_getWifiKey(DATABASE);
    document.getElementById('main-settings-modal').style.display='block';
    gui_dropdownMenuChangeStatus(); // Close menu
}
```

```
/**
 * Clears the add chat modal inputs and hides it.
 */
function gui_clearAddChatModal() {
    document.getElementById('add-chat-modal').style.display='none';
    document.getElementById('add-chat-input-title').value = "";
    document.getElementById('add-chat-input-id').value = "";
    document.getElementById('add-chat-input-key').value = "";
}

/**
 * Clears the chat settings modal inputs and hides it.
 */
function gui_clearChatSettingsModal() {
    document.getElementById('conv-settings-modal').style.display='none';
    document.getElementById('chat-settings-change-title').value = "";
    document.getElementById('chat-settings-change-key').value = "";
    document.getElementById('chat-settings-id-label').innerHTML = "";
}

/**
 * Clears the contacts modal and hides it.
 */
function gui_clearContactsModal() {
    document.getElementById('contacts-modal').style.display='none';
    document.getElementById('contact-list').innerHTML = "";
}

/**
 * Clears the add contact modal and hides it.
 */
function gui_clearAddOrChangeContactModal() {
    document.getElementById('add-change-contact-modal').style.display='none';
    document.getElementById('add-change-contact-input-name').value = "";
    var input = document.getElementById('add-change-contact-input-id');
    if(input != null) input.parentNode.removeChild(input);
    document.getElementById('add-change-contact-button').innerHTML = "";
}

/**
 * Clears the main settings modal inputs and hides it.
 */
function gui_clearMainSettingsModal() {
    document.getElementById('main-settings-modal').style.display='none';
    document.getElementById('main-settings-userid-input').value = "";
    document.getElementById('main-settings-username-input').value = "";
    document.getElementById('main-settings-ssid-input').value = "";
    document.getElementById('main-settings-key-input').value = "";
}

/**
```

```
 * Shows WebSocket modal.
 */
function gui_openWebSocketModal() {
    document.getElementById('websocket-modal').style.display='block';
}

/**
 * Hides WebSocket modal.
 */
function gui_closeWebSocketModal() {
    document.getElementById('websocket-modal').style.display='none';
}

/**
 * Adds a chat item to the sidebar list.
 * @param {int} id id of the chat
 * @param {string} name title of the chat
 */
function gui_addChatToSidebar(id, name) {
    var item = document.createElement("a");
    item.id = "chatlist-" + id;
    item.className = "w3-bar-item w3-button w3-black w3-hover-dark-gray";
    item.style = "white-space:nowrap; text-overflow:ellipsis;";
    item.addEventListener("click", function() { // Click listener
        if(ACTIVE_CHAT !== id) {
            gui_removeBubbleFromChat(id);
            gui_moveChatListItemFirst(id);
            gui_loadChat(id);
        }
        gui_sidebarClose();
    });
    var title = document.createElement("span");
    title.className = "w3-large";
    title.innerHTML = name;
    title.id = "chatlist-title-" + id;
    item.appendChild(title);
    item.appendChild(document.createElement("br"));
    document.getElementById("sidebar").appendChild(item);
}

/**
 * Adds a chat item to the sidebar, but with more details.
 * Useful for the first load.
 * @param {int} id chat id
 * @param {string} name chat title
 * @param {string} author last message author
 * @param {string} msg last message text
 * @param {int} unread unread counter
 */
function gui_addChatToSidebar2(id, name, author, msg, unread) {
    var item = document.createElement("a");
```

```
    item.id = "chatlist-" + id;
    item.className = "w3-bar-item w3-button w3-black w3-hover-dark-gray";
    item.style = "white-space:nowrap; text-overflow:ellipsis;";
    item.addEventListener("click", function() { // Click listener
        if(ACTIVE_CHAT !== id) {
            gui_removeBubbleFromChat(id);
            gui_moveChatListItemFirst(id);
            gui_loadChat(id);
        }
        gui_sidebarClose();
    });
    // Add title
    var title = document.createElement("span");
    title.className = "w3-large";
    title.innerHTML = name;
    title.id = "chatlist-title-" + id;
    item.appendChild(title);
    item.appendChild(document.createElement("br"));
    // Add last message
    var lm = document.createElement("span");
    lm.className = "w3-text-light-grey";
    if(author !== undefined || msg !== undefined)
        if(author == db_getUserId(DATABASE))
            lm.innerHTML = '(' + db_getUsername(DATABASE) + '): ' + msg;
        else
            lm.innerHTML = db_getContactName(DATABASE, author) + ': ' + msg;
    else
        lm.innerHTML = "";
    item.appendChild(lm);
    // Add bubble
    if(unread>0) {
        var b = document.createElement("span");
        b.className = "w3-tag w3-green w3-circle w3-right";
        b.innerHTML = unread;
        b.id = "bubble-" + id;
        item.appendChild(b);
    }
    document.getElementById("sidebar").appendChild(item);
}

/**
 * Removes a chat item from the sidebar list
 * @param {int} id chat id
 */
function gui_removeChatFromSidebar(id) {
    var chat = document.getElementById("chatlist-"+id);
    chat.parentNode.removeChild(chat);
}

/**
 * Moves a chat list item to the first place.
```

```
 * @param {int} id chat id
 */
function gui_moveChatListItemFirst(id){
    var parent = document.getElementById('sidebar');
    parent.insertBefore(document.getElementById('chatlist-'+id),
parent.childNodes[5]);
}

/**
 * Changes title of chat list item.
 * @param {int} id chat id
 * @param {string} title chat title
 */
function gui_changeChatListItemTitle(id, title){
    document.getElementById("chatlist-title-"+id).innerHTML = title;
}

/**
 * Adds a resume of the last message to the chat list item.
 * @param {int} id chat id
 * @param {string} msg last message text
 */
function gui_addLastMsgToChat(id, author, msg){
    var item = document.getElementById("chatlist-"+id);
    if(item.childNodes.length>2) item.removeChild(item.childNodes[2]); //
Clear
    var lm = document.createElement("span");
    lm.className = "w3-text-light-grey";
    if(author == db_getUserId(DATABASE))
        lm.innerHTML = '(' + db_getUsername() + '): ' + msg;
    else
        lm.innerHTML = db_getContactName(DATABASE, author) + ': ' + msg;
    document.getElementById("chatlist-"+id).appendChild(lm);
}

/**
 * Adds a counter of unread messages to a chat list item.
 * @param {int} id chat id
 * @param {int} num number of unread messages
 */
function gui_addBubbleToChat(id,num){
    gui_removeBubbleFromChat(id);
    var b = document.createElement("span");
    b.className = "w3-tag w3-green w3-circle w3-right";
    b.innerHTML = num;
    b.id = "bubble-" + id;
    document.getElementById("chatlist-"+id).appendChild(b);
}

/**
 * Removes unread messages counter from chat list item.
```

```
 * @param {int} id chat id
 */
function gui_removeBubbleFromChat(id){
    var b = document.getElementById("bubble-" + id);
    if(b != null) b.parentNode.removeChild(b);
}

/**
 * Adds an own message to the chatscreen.
 * @param {string} id message id (chat-author-msg)
 * @param {str} msg message text
 * @param {int} status message status (default: pending)
 */
function gui_addSentMsg(id, msg, status) {
    var row = document.createElement("div");
    row.className = "chat-screen-msg-row my-msg";
    row.id = "msg-" + id;
    var text = document.createElement("div");
    text.className = "chat-screen-msg-text";
    text.innerHTML = msg;
    var info = document.createElement("div");
    info.className = "chat-screen-msg-info";
    switch (status) {
        case MSG_PENDING: info.innerHTML = "sending..."; break;
        case MSG_SENT: info.innerHTML = "&#10555; sent"; break;
        case MSG_RECEIVED: info.innerHTML = "✓ received"; break;
        default: break;
    }
    row.appendChild(text);
    row.appendChild(info);
    document.getElementById("chat-msg-screen").appendChild(row);
    document.getElementById("chat-msg-screen").scrollTo(0,
document.getElementById("chat-msg-screen").scrollHeight);
}

/**
 * Adds a new remote message to the chat screen.
 * @param {string} id message id (chat-author-msg)
 * @param {int} sender author
 * @param {string} msg message text
 */
function gui_addReceivedMsg(id, sender, msg) {
    var row = document.createElement("div");
    row.className = "chat-screen-msg-row remote-msg";
    row.id = "msg-" + id;
    var text = document.createElement("div");
    text.className = "chat-screen-msg-text";
    text.innerHTML = '<b style="font-weight:bolder;">'+
db_getContactName(DATABASE, sender) + ': </b>' + msg;
    row.appendChild(text);
    document.getElementById("chat-msg-screen").appendChild(row);
```

```
    document.getElementById("chat-msg-screen").scrollTo(0,
document.getElementById("chat-msg-screen").scrollHeight);
}

/**
 * Cleans the chat input box.
 */
function gui_cleanChatInputBox() {
    document.getElementById("chat-input-box").value = "";
}

/**
 * Sends a LoRa text message:
 * 1- Creates the message and its ID.
 * 2- Sends it trough websocket.
 * 3- Saves it into the database.
 * 4- Does GUI stuff.
 */
function gui_sendMsg() {
    if(document.getElementById("chat-input-box").value != "" && ACTIVE_CHAT !=
undefined){
        if(db_getUserId(DATABASE) != undefined && db_getUserId(DATABASE) !=
"") {
            var text = document.getElementById("chat-input-box").value;
            var msgIdCount = db_getChatIdCounter(DATABASE, ACTIVE_CHAT) + 1;
// Create new message ID
            db_setChatIdCounter(DATABASE, ACTIVE_CHAT, msgIdCount); // Set the
new message ID in the database
            com_sendPacket(com_generateSendMessagePacket(ACTIVE_CHAT,
msgIdCount, text)); // Send packet
            db_addOwnMsgToChat(DATABASE, ACTIVE_CHAT, msgIdCount, text); //
Add message to database
            gui_addSentMsg(gen_generateMsgId(ACTIVE_CHAT,
db_getUserId(DATABASE), msgIdCount), text, MSG_PENDING);
            gui_addLastMsgToChat(ACTIVE_CHAT, db_getUsername(DATABASE), text);
            gui_moveChatListItemFirst(ACTIVE_CHAT);
            gui_cleanChatInputBox();
        } else {
            gui_alert("In order to send messages, you need to define your user
ID.", "red");
        }
    }
}

/**
 * Sends a message when the pressed key is an ENTER.
 * @param {event} e key event
 */
function gui_sendMsgOnEnter(e) {
    if (e.keyCode == 13) gui_sendMsg();
}
```

```
/**
 * Sets a message status if message's chat is active.
 * @param {int} chatId chat id
 * @param {string} msgId message id (chat-author-msg)
 * @param {int} status message status
 */
function gui_setMsgStatus(chatId, msgId, status) {
    if(ACTIVE_CHAT == chatId)
        switch (status) {
            case MSG_PENDING: document.getElementById("msg-
"+msgId).childNodes[1].innerHTML = "sending..."; break;
            case MSG_SENT: document.getElementById("msg-
"+msgId).childNodes[1].innerHTML = "&#10555; sent"; break;
            case MSG_RECEIVED: document.getElementById("msg-
"+msgId).childNodes[1].innerHTML = "✓ received"; break;
            default: break;
        }
}

/**
 * Loads a chat in the chatscreen.
 * @param {int} id chat id
 */
function gui_loadChat(id){
    ACTIVE_CHAT = id; // Set chat as active
    var chat = db_getChat(DATABASE, id);
    db_setChatUnreadCounter(DATABASE, id, 0);
    gui_setChatScreenTitle(chat.title)
    gui_clearChatScreen();
    for (var i = 0; i < chat.messages.length; i++){ // Load chats
        var msg = chat.messages[i];
        if(msg.mine) gui_addSentMsg(msg.id, msg.text, msg.status);
        else gui_addReceivedMsg(msg.id, db_getContactName(DATABASE,
msg.author), msg.text);
    }
    gui_showChatBox();
    if(document.getElementById("dropdown-active-chat-item") == null)
        gui_addActiveChatMenuItem();
    gui_sidebarClose();
    document.getElementById("chat-msg-screen").scrollTo(0,
document.getElementById("chat-msg-screen").scrollHeight);
}

/**
 * Adds a chat to the GUI and the database.
 */
function gui_addChat() {
    var title = document.getElementById('add-chat-input-title').value;
    var id = document.getElementById('add-chat-input-id').value;
    var key = document.getElementById('add-chat-input-key').value;
```

```
    if (title !== "" && id !== "" && key !== "") {
        db_addChat(DATABASE, Number(id), title, key);
        gui_addChatToSidebar(Number(id), title);
        gui_loadChat(Number(id));
        gui_clearAddChatModal();
        gui_alert("Chat created successfully!", "green");
    } else {
        gui_alert("Chat could not be created!", "red");
    }
}

/**
 * Changes active chat's title in database.
 */
function gui_changeChatTitle() {
    var title = document.getElementById('chat-settings-change-title').value;
    if (title !== "") {
        db_setChatTitle(DATABASE, ACTIVE_CHAT, title);
        gui_setChatScreenTitle(title);
        gui_changeChatListItemTitle(ACTIVE_CHAT, title);
        gui_alert("Chat title changed successfully!", "green");
    } else {
        gui_alert("Chat title could not be changed!", "red");
    }
}

/**
 * Changes active chat's key in database.
 */
function gui_changeChatKey() {
    var key = document.getElementById('chat-settings-change-key').value;
    if (key !== "") {
        db_setChatKey(DATABASE, ACTIVE_CHAT, key);
        gui_alert("Chat key changed successfully!", "green");
    } else {
        gui_alert("Chat key could not be changed!", "red");
    }
}

/**
 * Removes a chat from database and GUI.
 */
function gui_removeChat() {
    db_removeChat(DATABASE, ACTIVE_CHAT); // Remove chat from database
    gui_clearChatScreen(); // Clear chat screen
    gui_removeChatFromSidebar(ACTIVE_CHAT); // Remove from sidebar
    gui_removeActiveChatMenuItem(); // Remove active chat element from
dropdown
    gui_clearChatSettingsModal(); // Close modal
    gui_setChatScreenTitle(""); // Remove chat screen title
    ACTIVE_CHAT = undefined;
```

```
    gui_alert("Chat removed successfully!", "green");
}

/**
 * Changes user id in database.
 */
function gui_changeUserId() {
    var userId = document.getElementById('main-settings-userid-input').value;
    if (userId !== "") {
        db_setUserId(DATABASE, Number(userId));
        gui_alert("User ID changed successfully!", "green");
    } else {
        gui_alert("User ID could not be changed!", "red");
    }
}

/**
 * Changes username in the database.
 */
function gui_changeUserName() {
    var username = document.getElementById('main-settings-username-
input').value;
    if (username !== "") {
        db_setUsername(DATABASE, username);
        gui_alert("Username changed successfully!", "green");
    } else {
        gui_alert("Username could not be changed!", "red");
    }
}

/**
 * Changes wifi SSID in database.
 */
function gui_changeWifiSSID() {
    var ssid = document.getElementById('main-settings-ssid-input').value;
    if (ssid !== "") {
        db_setWifiSSID(DATABASE, ssid);
        gui_alert("Wifi SSID changed successfully!", "green");
    } else {
        gui_alert("Wifi SSID could not be changed!", "red");
    }
}

/**
 * Changes wifi key in database.
 */
function gui_changeWifiKey() {
    var key = document.getElementById('main-settings-key-input').value;
    if (key !== "") {
        db_setWifiKey(DATABASE, key);
        gui_alert("Wifi key changed successfully!", "green");
```

```
    } else {
        gui_alert("Wifi key could not be changed!", "red");
    }
}

/**
 * Adds a contact to the database.
 */
function gui_addContact() {
    var name = document.getElementById('add-change-contact-input-name').value;
    var id = document.getElementById('add-change-contact-input-id').value;
    if(id != '' && name != '') {
        if(!db_existsContact(DATABASE, Number(id))) { // If the contact is not
in the database
            db_addContact(DATABASE, Number(id), name);
            gui_clearAddOrChangeContactModal();
            gui_clearContactsModal();
            gui_openContactsModal();
            gui_dropdownMenuChangeStatus()
            gui_alert("Contact added!", "green");
        } else gui_alert("Contact already exists! Try again, please.", "red");
    } else gui_alert("Sorry. Invalid contact ID or contact name. Try again,
please.", "red");
}

/**
 * Changes a contact name in the database.
 * @param {string} contactId contact id
 */
function gui_changeContact(contactId) {
    var name = document.getElementById('add-change-contact-input-name').value;
    if(name != '') {
        db_setContactName(DATABASE, Number(contactId), name);
        gui_clearAddOrChangeContactModal();
        gui_clearContactsModal();
        gui_openContactsModal();
        gui_dropdownMenuChangeStatus();
        gui_alert("Contact changed!", "green");
    } else gui_alert("Sorry. Empty name not allowed. Try again, please.",
"red");
}

/**
 * Removes a contact from the database and the GUI.
 * @param {int} contactId
 */
function gui_removeContact(contactId) {
    var contact = document.getElementById('contact-list-'+contactId);
    if(contact != null) contact.parentNode.removeChild(contact);
    db_removeContact(DATABASE, contactId);
    gui_alert("Contact removed!", "green");
```

```
}

/**
 * Sends the database trough the websocket.
 */
function gui_saveDatabase() {
    com_sendPacket(com_generateSendDatabasePacket());
    gui_dropdownMenuChangeStatus();
}
```